Exhibit 2

microunity

Terpsichore System Architecture

Copy Number: 247 **REDACTED**

Issued To: _____
Final Test

Issued By: _____
(MicroUnity officer or director)

Craig Hansen
Chief Architect
MicroUnity Systems Engineering, Inc.
255 Caspian Drive
Sunnyvale, CA 94089-1015
Tel: (408) 734-8100 Fax: (408) 734-8136
EMail: craig@microunity.com

Contents

MU 0023214

Tables

Figures

Introduction

MicroUnity's Terpsichore System Architecture describes general-purpose processor, memory, and interface subsystems, organized to operate at enormously higher bandwidth rates than traditional computers.

Terpsichore's Euterpe processor performs integer, floating point, and signal processing operations at data rates up to 512 bits (i.e., up to four 128-bit operand groups) per instruction. The instruction set design carries the concept of streamlining beyond Reduced Instruction Set Computer (RISC) architectures, since it targets implementations that issue several instructions per machine cycle.

The Terpsichore memory subsystem provides 32/64/128-bit virtual addressing and 64-bit physical addressing for UNIX, Mach, and other advanced OS environments. Caches supply the high data and instruction issue rates of the processor, and support coherency primitives for scalable multiprocessors. The memory subsystem includes mechanisms for sustaining high data rates not only in block transfer modes, but also in non-unit stride and scatter/gather access patterns.

Hermes channels provide 64-bit transfers between subsystem components with gigabyte-per-second bandwidth. Terpsichore's Cerberus serial bus provides a flexible, robust and inexpensive mechanism to handle system initialization, configuration, availability, and error recovery. Mnemosyne memory interface devices provide for the integration of large numbers of industry-standard memory components into Terpsichore systems.

Terpsichore's Calliope interface subsystem is tightly integrated with the processor and memory, to supply both the bandwidth and real-time response needs of video, audio, network, and mass storage interfaces. Integration provides for the sharing of memory bandwidth among these devices and the processor, without distributed or dedicated buffer memories in each interface adapter.

Terpsichore's Euterpe processor incorporates Icarus interprocessor interfaces for assembly of small-scale, coherently-cached, shared-memory multiprocessors, without additional circuitry. Icarus interfaces may also be used to connect Terpsichore processors to a high-performance switching fabric for large-scale multiprocessors, or to adapters to standard interprocessor interfaces, such as Scalable Coherent Interface (IEEE standard 1596-1992).

The goal of the Terpsichore architecture is to integrate these processor, memory, and interface capabilities with optimal simplicity and generality. From the software perspective, the entire machine state consists of a program counter, a single bank of 64 general-purpose 64-bit registers, and a linear byte-addressed shared memory space with mapped interface registers. All interrupts and exceptions are precise, and occur with such low overhead that even cache misses can be handled as exceptions under software control.

This document is intended for Terpsichore software and hardware developers alike, and defines the interface at which their designs must meet. Terpsichore

pursues the most efficient tradeoffs between hardware and software complexity by making all processor, memory, and interface resources directly accessible to high-level language programs.

Conformance

To ensure that Terpsichore systems are able to freely interchange data, user-level programs, system-level programs and interface devices, the Terpsichore system architecture reaches above the processor level architecture.

Mandatory and Optional Areas

A computer system conforms to the requirements of the Terpsichore System Architecture if and only if it implements all the specifications described in this document and other specifications included by reference. Conformance to the specification is mandatory in all areas, including the instruction set, memory management system, interface devices and external interfaces, and bootstrap ROM functional requirements, except where explicit options are stated.

Optional areas include:

- Number of processor threads
- Size of first-level cache memories
- Existence of a second-level cache
- Size of second-level cache memory
- Size of system-level memory
- Existence of certain optional interface device interfaces

Conformance to the specification is also optional regarding the physical implementation of internal interfaces, specifically that of the Cerberus serial bus architecture, the Hermes high-bandwidth channel architecture, and the Icarus interprocessor interconnection architecture. An implementation may replace, modify or eliminate these interfaces, provided that the software-level functionality is unchanged.

Upward-compatible Modifications

From time to time, MicroUnity may modify the architecture in an upward-compatible manner, such as by the addition of new instructions, definition of reserved bits in system state, or addition of new standard interfaces. Such modifications will be added as options, so that designs which conform to this version of the architecture will conform to future, modified versions.

Additional devices and interfaces, not covered by this standard may be added in specified regions of the physical memory space, provided that system reset places these devices and interfaces in an inactive state that does not interfere with the operation of software that runs in any conformant system. The software interface requirements of any such additional devices and interfaces must be made as widely available as this architecture specification.

Unrestricted Physical Implementation

Nothing in this specification should be construed to limit the implementation choices of the conformant system beyond the specific requirements stated herein. In particular, a computer system may conform to the Terpsichore System Architecture while employing any number of components, dissipate any amount of heat, require any special environmental facilities, or be of any physical size.

Draft Version

This document is a draft version of the architectural specification. In this form, conformance to this document may not be claimed or implied. MicroUnity may change this specification at any time, in any manner, until it has been declared final. When this document has been declared final, the only changes will be to correct bugs, defects or deficiencies, and to add upward-compatible optional extensions.

Overview

Notation

The descriptive notation used in this document is summarized in the table below:

x + y	two's complement or floating-point addition of x and y
x - y	two's complement or floating-point subtraction of y from x
x * y	two's complement or floating-point multiplication of x and y
x / y	two's complement or floating-point division of x by y
x = y	two's complement or floating-point equality comparison between x and y. Result is a single bit.
x ≠ y	two's complement or floating-point inequality comparison between x and y. Result is a single bit.
x < y	two's complement or floating-point less than comparison between x and y. Result is a single bit.
x ≥ y	two's complement or floating-point greater than or equal comparison between x and y. Result is a single bit.
$\sqrt{x}$	floating-point square root of x
x ‖ y	concatenation of bit field x to left of bit field y
x^y	binary digit x repeated or concatenated y times
x_y	extraction of bit y (using little-endian bit numbering) from value x
$x_{y..z}$	extraction of bit field formed from bits y through z of value x
x?y:z	value of y, if x is true, otherwise value of z. Value of x is a single bit.
x ← y	bitwise assignment of x to value of y
Sn	signed two's complement, binary data format of n bytes
Un	unsigned binary data format of n bytes
Fn	floating-point data format of n bytes

descriptive notation

The ordering of bits in this document is always little-endian, regardless of the ordering of bytes within larger structures.

Instruction mnemonics are usually written with periods (.) separating elements of the mnemonic to make them easier to understand. Terpsichore assemblers and other code tools treat these periods as optional; the mnemonics are designed to be parsed either with or without the periods.

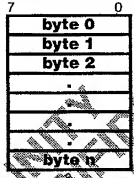
Representation

Terpsichore memory is byte-addressed, using either little-endian or big-endian byte ordering. The selection of byte ordering is dynamic, so that little-endian and big-endian processes, and even data structures within a process, can be intermixed on the processor. Terpsichore provides eight-byte (64-bit) virtual

address, physical address, and data path sizes, and uses fixed-length four-byte (32-bit) instructions. Arithmetic is performed on two's-complement or unsigned binary and ANSI/IEEE standard 754-1985 conforming binary floating-point number representations.

Memory

Memory is an array of bytes, without a specified byte ordering.



Memory read/load semantics

Terpsichore memory, including memory-mapped registers, must conform to the following requirements regarding side-effects of read or load operations:

A memory read must have no side-effects on the contents of the addressed memory nor on the contents of any other memory.

Memory write/store semantics

Terpsichore memory, including memory-mapped registers, must conform to the following requirements regarding side-effects of read or load operations:

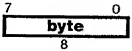
A memory write must have no side-effects on the contents of the addressed memory. A memory write may cause side-effects on the contents of memory not addressed by the write operation, however, a second memory write of the same value to the same address must have no side-effects on any memory; memory write operations must be idempotent.

Euterpe store instructions which are weakly ordered may have side-effects on the contents of memory not addressed by the store itself; subsequent load instructions which are also weakly ordered may or may not return values which reflect the side-effects.

Fixed-point Data

Byte

A byte is a single element of the memory array:



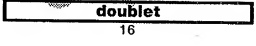
Larger data structures are constructed from the concatenation of bytes in either little-endian or big-endian byte ordering. A memory access of a data structure of size s at address i is formed from memory bytes at addresses i through i+s-1. Unless otherwise specified, there is no specific requirement of alignment: it is not generally required that i be a multiple of s. Aligned accesses are preferred whenever possible, however, as they will often require one less processor or memory clock cycle than unaligned accesses.

With little-endian byte ordering, the bytes are arranged as:

With big-endian byte ordering, the bytes are arranged as:

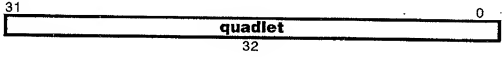
Doublet

A doublet is the concatenation of two bytes:



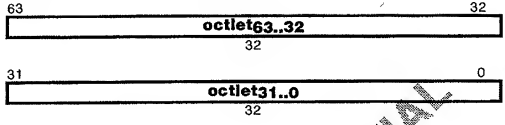
Quadlet

A quadlet is the concatenation of four bytes:



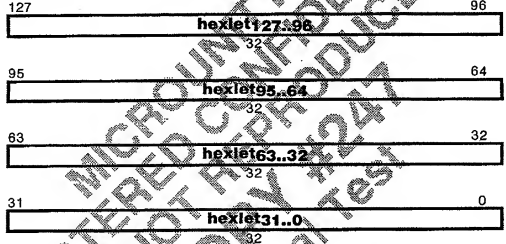
Octlet

An octlet is the concatenation of eight bytes:



Hexlet

A hexlet is the concatenation of sixteen bytes:



Address

Terpsichore addresses are quadlet, octlet, or hexlet quantities, depending on the level of the implementation.

Floating-point Data

MU 0023230

Terpsichore's floating-point formats are designed to satisfy ANSI/IEEE standard 754-1985: Binary Floating-point Arithmetic. Standard 754 leaves certain aspects to the discretion of the implementor:

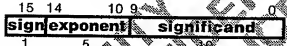
Terpsichore adds additional half-precision and quad-precision formats to standard 754's single-precision and double-precision formats. Terpsichore's double-precision satisfies standard 754's precision requirements for a single-extended format, and Terpsichore's quad-precision satisfies standard 754's precision requirements for a double-extended format.

Quiet NaN values are denoted by any sign bit value, an exponent field of all one bits, and a non-zero significand with the most significant bit cleared. Quiet NaN values generated by default exception handling of standard operations have a zero sign bit, an exponent field of all one bits, and a significand field with the most significant bit cleared, the next-most significant bit set, and all other bits cleared.

Signaling NaN values are denoted by any sign bit value, an exponent field of all one bits, and a non-zero significand with the most significant bit set.

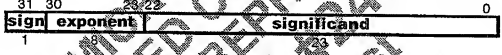
Half-precision Floating-point

Terpsichore half precision uses a format similar to standard 754's requirements, reduced to a 16-bit overall format. The format contains sufficient precision and exponent range to hold a 12-bit signed integer.



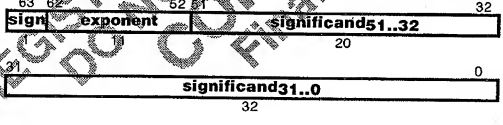
Single-precision Floating-point

Terpsichore single precision satisfies standard 754's requirements for "single."



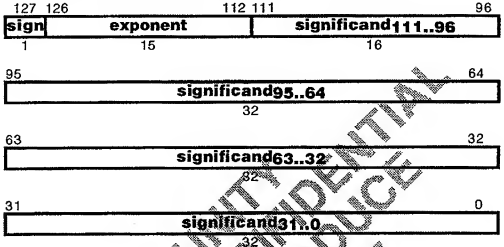
Double-precision Floating-point

Terpsichore double precision satisfies standard 754's requirements for "double."



Quad-precision Floating-point

Terpsichore quad precision satisfies standard 754's requirements for "double extended," but has additional significand precision to use 128 bits.

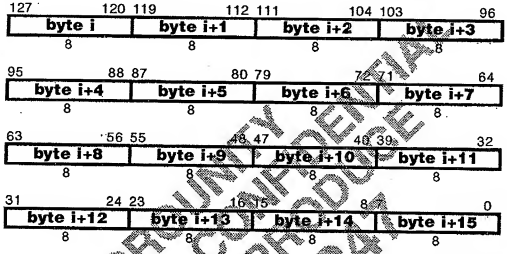


Instruction

A Terpsichore instruction is specifically defined as a four-byte structure with the ordering shown below. It is different from the quadlet defined above because the placement of instructions into memory must be independent of the byte ordering used for data structures. Instructions must be aligned on four-byte boundaries: in the diagram below, i must be a multiple of 4.

Gateway

A Terpsichore gateway is specifically defined as a 16-byte structure with the ordering shown below. A gateway contains a code address used to invoke a procedure at a higher privilege level securely. Gateways are marked by protection information specified in the TLB. Gateways must be aligned on 16-byte boundaries, that is, in the diagram below, i must be a multiple of 16.



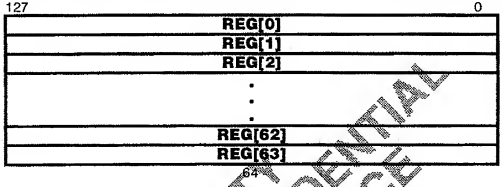
The gateway contains a code address, right-aligned within the 128 bit structure:

User state

The user state consists of hardware data structures that are accessible to all conventional compiled code. The Terpsichore user state is designed to be as regular as possible, and consists only of the general registers, the program counter, and virtual memory. There are no specialized registers for condition codes, operating modes, rounding modes, integer multiply/divide, or floating-point values.

General Registers

Terpsichore user state includes 64 general registers. All are identical; there is no dedicated zero-valued register, and there are no dedicated floating-point registers.



Program Counter

The program counter contains the address of the currently executing instruction. This register is implicitly manipulated by branch instructions, and read by branch instructions that save a return address in a general register.

The program counter may be implemented as 30, 62, or 126 bits, depending upon the level of implementation. Unimplemented bits are always zero.

Privilege Level

The privilege level register contains the privilege level of the currently executing instruction. This register is implicitly manipulated by branch gateway and branch down instructions, and read by branch gateway instructions that save a return address in a general register.

System state

The system state consists of the facilities not normally used by conventional compiled code. These facilities provide mechanisms to execute such code in a fully virtual environment. All system state is memory mapped, so that it can be manipulated by compiled code.

Fixed-point

Terpsichore provides load and store instructions to move data between memory and the registers, branch instructions to compare the contents of registers and to transfer control from one code address to another, and arithmetic operations to perform computation on the contents of registers, returning the result to registers.

Load and Store

The load and store instructions move data between memory and the registers. When loading data from memory into a register, values are zero-extended or sign-extended to fill the register. When storing data from a register into memory, values are truncated on the left to fit the specified memory region.

Load and store instructions that specify a memory region of more than one byte may use either little-endian or big-endian byte ordering; the size and ordering are explicitly specified in the instruction. Regions larger than one byte may be either aligned to addresses that are an even multiple of the size of the region, or of unspecified alignment: alignment checking is also explicitly specified in the instruction.

The load and store instructions are used for fixed-point data as well as floating-point and digital signal processing data; Terpsichore has a single bank of registers for all data types.

Swap instructions provide multithread and multiprocessor synchronization, using indivisable operations: add-and-swap, compare-and-swap, and multiplex-and-swap. A store-multiplex operation provides the ability to indivisably write to a portion of an octlet. These instructions always operate on aligned octlet data, using either little-endian or big-endian byte ordering.

Branch Conditionally

The fixed-point compare-and-branch instructions provide all arithmetic tests for equality and inequality of signed and unsigned fixed-point values. Tests are performed either between two operands contained in general registers, or on the bitwise and of two operands. Depending on the result of the compare, either a branch is taken, or not taken. A taken branch causes an immediate transfer of the program counter to the target of the branch, specified by a 12-bit signed offset from the location of the branch instruction. A non-taken branch causes no transfer; execution continues with the following instruction.

MU 0023235

Branch Unconditionally

Other branch instructions provide for unconditional transfer of control to addresses too distant to be reached by a 12-bit offset, and to transfer to a target while placing the location following the branch into a register. The branch through gateway instruction provides a secure means to access code at a higher privilege level, in a form similar to a normal procedure call.

Highly Confidential

Arithmetic Operations

The fixed-point arithmetic operations include add, subtract, multiply, divide, shifts, and set on compare, all using octlet-sized operands. Multiply and divide operations produce hexlet results; all other operations produce octlet results.

When specified, add, subtract, and shift operations may cause a fixed-point arithmetic exception to occur on resulting conditions such as signed overflow, or signed or unsigned equality or inequality to zero.

Floating-point

Terpsichore provides all the facilities mandated and recommended by ANSI/IEEE standard 754-1985: Binary Floating-point Arithmetic, with the use of supporting software.

Branch Conditionally

The floating-point compare-and-branch instructions provide all the comparison types required and suggested by the IEEE floating-point standard. These floating-point comparisons augment the usual types of numeric value comparisons with special handling for NaN (not-a-number) values. A NaN compares as "unordered" with respect to any other value, even that of an identical NaN.

Terpsichore floating-point compare-and-branch instructions do not generate an exception on comparisons involving quiet or signaling NaNs; if such exceptions are desired, they can be obtained by combining the use of a floating-point compare and set instruction, with either a floating-point compare-and-branch instruction on the floating-point operands or a fixed-point compare-and-branch on the set result.

Because the less and greater relations are anti-commutative, one of each relation that differs from another only by the replacement of an L with a G in the code can be removed by reversing the order of the operands and using the other code. Thus, a UL relation can be used in place of a UG relation by swapping the operands to the compare-and-branch or compare-and-set instruction.

The E and NE relations can be used to determine the unordered condition of a single operand by comparing the operand with itself.

The following floating-point compare-and-branch relations are provided:

Mnemonic		Branch taken if values compare as:				Exception if	
code	C-like	Unord-ered	Greater	Less	Equal	unord-ered	invalid
E	==	F	F	F	T	no	no
NE	!=	T	T	T	F	no	no
UE	?=	T	F	F	T	no	no
NUE	!?=	F	T	T	F	no	no
NUGE	!?>=	F	F	T	F	no	no
UGE	?>=	T	T	F	T	no	no
UL	?<	T	F	T	F	no	no
NUL	!?<	F	T	F	F	no	no

compare-and-branch relations

Compare-and-set

The floating-point compare-and-set instructions provide all the comparison types supported as compare-and-branch instructions. Terpsichore floating-point compare-and-set instructions may optionally generate an exception on comparisons involving quiet or signaling NaNs.

The following floating-point compare-and-branch relations are provided:

Mnemonic		Result if values compare as:				Exception if	
code	C-like	Unord-ered	Greater	Less	Equal	unord-ered	invalid
E	==	F	F	F	T	no	no
NE	!=	T	T	T	F	no	no
UE	?=	T	F	F	T	no	no
NUE	!?=	F	T	T	F	no	no
NUGE	!?>=	F	F	T	F	no	no
UGE	?>=	T	T	F	T	no	no
UL	?<	T	F	T	F	no	no
NUL	!?<	F	T	F	T	no	no
E.X	==	F	F	F	T	no	yes
NE.X	!=	T	T	T	F	no	yes
UE.X	?=	T	F	F	T	no	yes
NUE.X	!?=	F	T	T	F	no	yes
L.X	<	F	F	T	F	yes	yes
NL.X	!<	T	T	F	T	yes	yes
NGE.X	!>=	T	F	T	F	yes	yes
GE.X	<=	F	T	F	T	yes	yes

compare-and-branch relations

Arithmetic Operations

The operations supported in hardware are floating-point add, subtract, multiply, divide, and square root. Other operations required by the ANSI/IEEE floating-point standard are provided by software libraries.

The operations explicitly specify the precision of the operation, and round the result to the specified precision at the conclusion of each operation.

A single instruction provides a floating-point multiply with the result fed into a floating-point add. The result is computed as if the multiply is performed to infinite precision, added as if in infinite precision, then rounded. This operation is a particularly good match to the needs of vector linear algebra routines.

Rounding

Rounding is specified within the instructions explicitly, to avoid maintaining explicit state for a rounding mode.

Exceptions

All the mandated floating-point exception conditions cause a trap when they occur; maintenance of sticky and other status bits is performed using software routines. Because the floating-point inexact exception may be very frequent, this exception only occurs when specified in the instruction explicitly. Arithmetic operations may also specify that all exceptions are to be handled by default, generating special results instead of traps.

Digital Signal Processing

The Terpsichore processor provides a set of operations that maintain the fullest possible use of 64- and 128-bit data paths when operating on lower-precision fixed-point or floating-point vector values. These operations are useful for several application areas, including digital signal processing, image processing, and synthetic graphics. The basic goal of these operations is to accelerate the performance of algorithms that exhibit the following characteristics:

Low-precision arithmetic

The operands and intermediate results are fixed-point values represented in no greater than 64 bit precision. For floating-point arithmetic, operands and intermediate results are of 16, 32, or 64 bit precision.

The use of fixed-point arithmetic permits various forms of operation reordering that are not permitted in floating-point arithmetic. Specifically, commutativity and associativity, and distribution identities can be used to reorder operations. Compilers can evaluate operations to determine what intermediate precision is required to get the specified arithmetic result.

Terpsichore supports several levels of precision, as well as operations to convert between these different levels. These precision levels are always powers of two, and are explicitly specified in the operation code.

Sequential access to data

The algorithms are or can be expressed as operations on sequentially ordered items in memory. Scatter-gather memory access or sparse-matrix techniques are not required.

Where an index variable is used with a multiplier, such multipliers must be powers of two. When the index is of the form: nx+k, the value of n must be a power of two, and the values referenced should have k include the majority of values in the range 0..n-1. A negative multiplier may also be used.

Vectorizable operations

The operations performed on these sequentially ordered items are identical and independent. Conditional operations are either rewritten to use boolean variables or masking, or the compiler is permitted to convert the code into such a form.

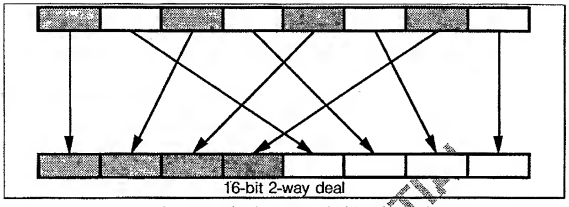
Data-handling Operations

The characteristics of these algorithms include sequential access to data, which permit the use of the normal load and store operations to reference the data. Octlet and hexlet loads and stores reference several sequential items of data, the number depending on the operand precision.

The discussion of these operations is independent of byte ordering, though the ordering of bit fields within octlets and hexlets must be consistent with the ordering used for bytes. Specifically, if big-endian byte ordering is used for the loads and stores, the figures below should assume that index values increase from left to right, and for little-endian byte ordering, the index values increase from right to left. For this reason, the figures indicate different index values with different shades, rather than numbering.

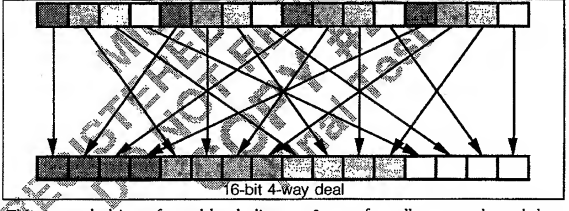
When an index of the nx+k form is used in array operands, where n is a power of 2, data memory sequentially loaded contains elements useful for separate operands. The "deal" instruction divides a hexlet of data up into two octlets, with alternate bit fields of the source hexlet grouped together into the two results. For example, a G.DEAL.16 operation rearranges the source hexlet into two octlets as follows:[1]

[1] An example of the use of a deal can be found in the appendix: Digital Signal Processing Applications: Decimation of Monochrome Image or Decimation of Color Image



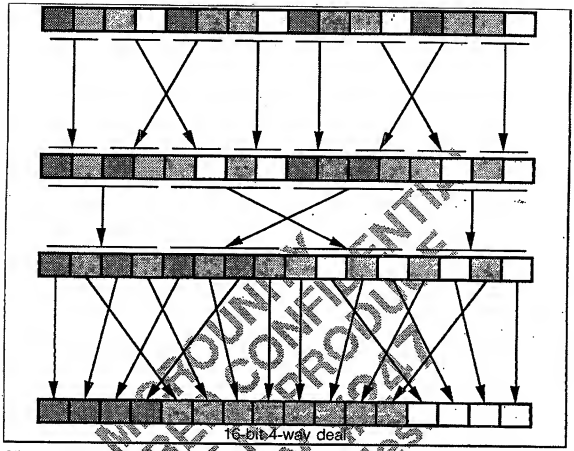
In the deal operation, the source hexlet is specified by two octlet registers, and the two result octlets are specified as a hexlet register pair. (This sounds backwards, and it really is, but it works in practice, because the result is usually used in operations that accept octlet operands. Ideally, the source hexlet should be a register pair, and the result should be two octlet registers.)

The example above directly applies to the case where n is 2. When n is larger, a series of DEAL operations can be used to further subdivide the sequential stream. For example, when n is 4, we need to deal out 4 sets of doublet operands, as shown in the figure below:[2]



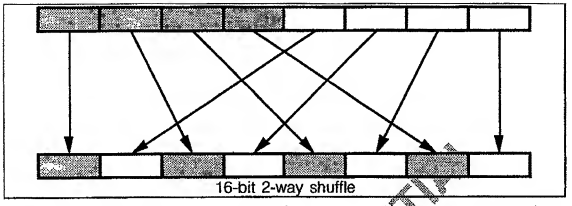
This 4-way deal is performed by dealing out 2 sets of quadlet operands, and then dealing each of them out into 2 sets of doublet operands.

[2]An example of the use of a four-way deal can be found in the appendix: Digital Signal Processing Applications: Conversion of Color to Monochrome

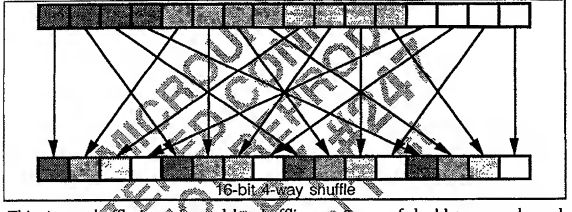


There are three rows of arrows shown above. The first row is the result of two G.DEAL.32 operations, each independently dealing 2 sets of pairs of doublets. The result of these two operations is the second row of boxes. The last row is the result of two independent G.DEAL.16 operations, each dealing 2 sets of doublets into register pairs. The middle row of arrows shows the implicit action performed by specifying two non-adjacent registers for the hexlet sources of the G.DEAL.16 operations.

When an array result of computation is accessed with an index of the form nx+k, for n a power of 2, the reverse of the "deal" operation needs to be performed on vectors of results to interleave them for storage in sequential order. The "shuffle" operation interleaves the bit fields of two octlets of results into a single hexlet. For example a G.SHUFFLE.16 operation combines two octlets of doublet fields into a hexlet as follows:

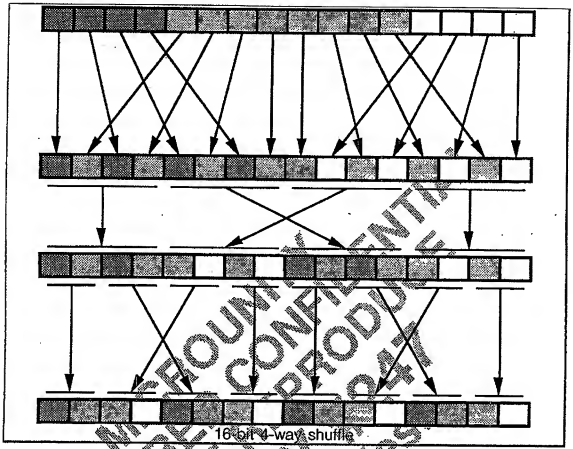


For larger values of n, a series of shuffle operations can be used to combine additional sets of fields, similarly to the mechanism used for the deal operations. For example, when n is 4, we need to shuffle up 4 sets of doublet operands, as shown in the figure below:[3]



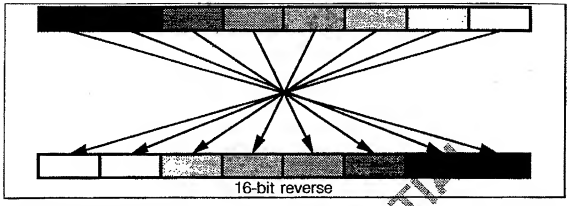
This 4-way shuffle is performed by shuffling up 2 sets of doublet operands, and then shuffling each of them up as 2 sets of quadlet operands.

[3]An example of the use of a four-way shuffle can be found in the appendix: Digital Signal Processing Applications: Conversion of Monochrome to Color



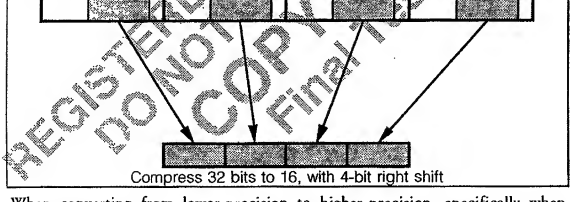
There are three rows of arrows shown above. The first row is the result of two G.SHUFFLE.16 operations, each independently shuffling 2 sets of pairs of doublets. The result of these two operations is the second row of boxes. The last row is the result of two independent G.SHUFFLE.32 operations, each shuffling 2 sets of quadlets into register pairs. The middle row of arrows shows the implicit action performed by specifying two non-adjacent registers for the two octlet sources of the G.SHUFFLE.32 operations.

When the index of a source array operand or a destination array result is negated, or, in other words, if of the form nx+k where n is negative, the elements of the array must be arranged in reverse order. The "swap" operation reverses the order of the bit fields in a hexlet. For example, a G.SWAP.16 operation reverses the doublets within a hexlet:



Variations of the deal and shuffle operations are also useful for converting from one precision to another. This may be required if one operand is represented in a different precision than another operand or the result, or if computation must be performed with intermediate precision greater than that of the operands, such as when using an integer multiply.

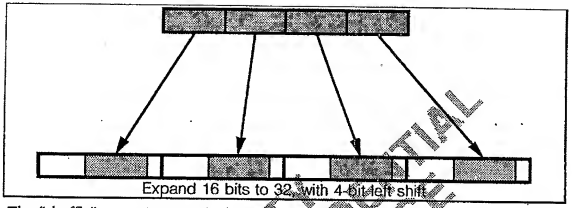
When converting from a higher precision to a lower precision, specifically when halving the precision of a hexlet of bit fields, half of the data must be discarded, and the bit fields packed together. The "compress" operation is a variant of the "deal" operation, in which the operand is a hexlet, and the result is an octlet. An arbitrary half-sized sub-field of each bit field can be selected to appear in the result. For example, a selection of bits 19..4 of each quadlet in a hexlet is performed by the G.COMPRESS.16.4 operation:



When converting from lower-precision to higher-precision, specifically when doubling the precision of an octlet of bit fields, one of several techniques can be used, either multiply, expand, or shuffle. Each has certain useful properties. In the discussion below, m is the precision of the source operand.

The multiply operation, described in detail below, automatically doubles the precision of the result, so multiplication by a constant vector will simultaneously double the precision of the operand and multiply by a constant that can be represented in m bits.

An operand can be doubled in precision and shifted left with the "expand" operation, which is essentially the reverse of the "compress" operation. For example the G.EXPAND.16,4 expands from 16 bits to 32, and shifts 4 bits left:



The "shuffle" operation can double the precision of an operand and multiply it by 1 (unsigned only), 2^m or 2^m+1, by specifying the sources of the shuffle operation to be a zeroed register and the source operand, the source operand and zero, or both to be the source operand. When multiplying by 2^m, a constant can be freely added to the source operand by specifying the constant as the right operand to the shuffle.

Arithmetic Operations

The characteristics of the algorithms that affect the arithmetic operations most directly are low-precision arithmetic, and vectorizable operations. The fixed-point arithmetic operations provided are most of the functions provided in the standard integer unit, except for those that check conditions. These functions include add, subtract, bitwise boolean operations, shift, set on condition, and multiply, in forms that take packed sets of bit fields of a specified size as operands. The floating-point arithmetic operations provided are as complete as the scalar floating-point arithmetic set. The result is generally a packed set of bit fields of the same size as the operands, except that the fixed-point multiply function intrinsically doubles the precision of the bit field.

Conditional operations are provided only in the sense that the set on condition operations can be used to construct bit masks that can select between alternate vector expressions, using the bitwise boolean operations. All instructions operate over the entire octlet or hexlet operands, and produce a hexlet result. The sizes of the bit fields supported are always powers of two.

Galois Field Operations

MU 0023245

Terpsichore provides a general software solution to the most common operations required for Galois Field arithmetic. The instruction provided is a polynomial divide, with the polynomial specified as one register operand. The result of a specified number of division steps, expressed as a register pair, is the result of the instruction. This instruction can be used to perform CRC generation and

checking, Reed-Solomon code generation and checking, and spread-spectrum encoding and decoding.

Register Usage

All Terpsichore registers are identical and general-purpose; there is no dedicated zero-valued register, and no dedicated floating-point registers. By software convention, the non-specific general registers are used in more specific ways.

register number	usage	how saved
0	link	caller
1	dp	caller
2-9	parameters	caller
10-31	temporary	caller
32-61	saved	callee
62	fp, when required	callee
63	sp	callee

register usage

At a procedure call boundary, registers are saved either by the caller or callee procedure, which provides a mechanism for leaf procedures to avoid needing to save registers. Optimizers may choose to allocate variables into caller or callee saved registers depending on how their lifetimes overlap with procedure calls.

The dp register points to a small (4 kilobyte) array of pointers, literals, and statically-allocated variables which is used locally to the procedure. The uses of the dp register are similar to the Mips use of the gp register, except that each procedure may have a different value, which expands the space addressable by small offsets from this pointer. This is an important distinction, as the offset field of Terpsichore load and store instructions are only 12 bits. The compiler may use additional registers and/or indirect pointers to address larger regions.

This mechanism also permits code to be shared, with each static instance of the dp region assigned to a different address in memory. In conjunction with position-independent or pc-relative branches, this allows library code to be dynamically relocated and shared between processes.

MU 0023246

Procedure Calling Conventions

Procedure parameters are normally allocated in registers, starting from register 2 up to register 9. These registers hold up to 8 parameters, which may each be of any size from one byte to eight bytes, including single-precision and double-precision floating-point parameters. Quad-precision floating-point parameters require an aligned pair of registers. The C varargs.h or stdarg.h conventions may require saving registers into memory (this is not necessarily so, but some semi-portable semi-conventions such as _doprnt would break otherwise). Procedure

return values are also allocated in registers, starting from register 2 up to register 9.

There are several data structures maintained in registers for the procedure calling conventions: link, sp, dp, fp. The link register contains the address to which the callee should return to at the conclusion of the procedure.

The sp register is used to form addresses to save parameter and other registers, maintain local variables, i.e., data that is allocated as a LIFO stack. For procedures that require a stack, normally a single allocation is performed, which allocates space for input parameters, local variables, saved registers, and output parameters all at once. The sp register is always 16-byte aligned.

The dp register is used to address pointers, literals and static variables for the procedure. The newpc register is loaded with the entry point of the procedure, and the newdp register is loaded with the value of the dp register required for the procedure. This mechanism provides for dynamic linking, by initially filling in the link and dp fields in the data structure to point to the dynamic linker. The linker can use the current contents of the link and/or dp registers to determine the identity of the caller and callee, to find the location to fill in the pointers and resume execution.

The fp register is used to address the stack frame when the stack size varies during execution of a procedure, such as when using the GNU alloca function. When the stack size can be determined at compile time, the sp register is used to address the stack frame and the fp register may be used for other general purposes as a callee-saved register.

Typical static-linked, intra-module calling sequence:

caller or callee (non-leaf):

```
    A.ADDI      sp,size
    S.64        link,off(sp)
    ... (using same dp as caller)
    B.LINK.I    callee
    ...
    L.64        link,off(sp)
    A.ADDI      sp,size
    B           link
```

callee (leaf):

```
    ... (using dp)
    B           link
```

Typical dynamic-linked, inter-module calling sequence:

caller or callee (non-leaf):

```
    A.ADDI      sp,-size
    S.64        link,off(sp)
    S.64        dp,off(sp)
    ... (using dp)
    L.64        link,off(dp)
```

```
L.64        dp,off(dp)
B.LINK      link,link
L.64        dp,off(sp)
... (using dp)
L.64        link,off(sp)
A.ADDI      sp,size
B           link
```

callee (leaf):

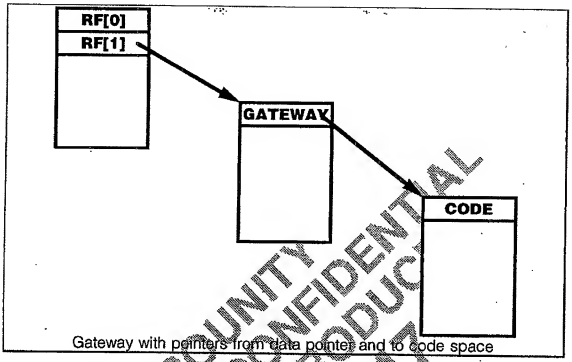
```
... (using dp)
B           link
```

The load instruction is required in the caller following the procedure call to restore the dp register. A second load instruction also restores the link register, which may be located at any point between the last procedure call and the branch instruction which returns from the procedure.

System and Privileged Library Calls

It is an objective to make calls to system facilities and privileged libraries as similar as possible to normal procedure calls as described above. Rather than invoke system calls as an exception, which involves significant latency and complication, we prefer to use a modified procedure call in which the process privilege level is quietly raised to the required level. In to provide this mechanism safely, interaction with the virtual memory system is required.

Such a routine must not be entered from anywhere other than its legitimate entry point, otherwise the sudden access to a higher privilege level might be taken advantage of. The branch-gateway instruction ensures both that the procedure is invoked at a proper entry point, and that the data pointer is properly set. To ensure this, the branch-through-gateway instruction retrieves a "gateway" directly from the protected virtual memory space. The gateway is accessed via the data pointer, and contains the virtual address of the entry point of the procedure. A gateway can only exist in regions of the virtual address space designated to contain them, to ensure that a gateway cannot be forged.



Similarly, a return from a system or privileged routine involves a reduction of privilege. This need not be carefully controlled by architectural facilities, so a procedure may freely branch to a less-privileged code address. However, in certain, though perhaps rare, cases it would be useful to have highly privileged code call less-privileged routines. As an example, a user may request that errors in a privileged routine be reported by invoking a user-supplied error-logging routine. In such a case, a return from a procedure actually requires an increase in privilege, which must be carefully controlled. Again, a branch-through-gateway instruction can be used, this time in the instruction following the call, to raise the privilege again in a controlled fashion. In such a case, special care must be taken to ensure that the less-privileged routine is not permitted to gain unauthorized access by corruption of the stack or saved registers, such as by saving all registers and setting up a new stack frame that may be manipulated by the less-privileged routine.

Typical dynamic-linked, inter-gateway calling sequence:

caller:

```
...
S.64        link,off(sp)
S.64        dp,off(sp)
...
L.64        dp,off(dp)
B.GATE      link,off(dp)
L.64        link,off(sp)
L.64        dp,off(sp)
```

callee (non-leaf):

```
S.64          sp,off(dp)
L.64          sp,off(dp)
S.64          link,off(sp)
S.64          dp,off(sp)
... (using dp)
L.64          link,off(sp)
L.64          dp,off(sp)
L.64          sp,off(dp)
B.DOWN        link
```

callee (leaf):

```
... (using dp)
B.DOWN        link
```

The callee, if it uses a stack for local variable allocation, cannot necessarily trust the value of the sp passed to it, except as a region to receive parameters held in memory.

Pipeline Organization

Terpsichore performs all instructions as if executed one-by-one, in-order, with precise exceptions always available. Consequently, code which ignores the subsequent discussion of Terpsichore pipeline implementations will still perform correctly. However, the highest performance of the Terpsichore processor is achieved only by matching the ordering of instructions to the characteristics of the pipeline. In the following discussion, the general characteristics of all Terpsichore implementations precedes discussion of specific choices for specific implementations.

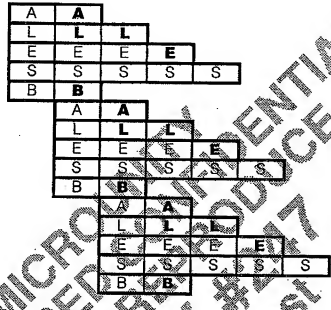
Super-string Pipeline

Terpsichore is designed to fetch and execute several instructions in each clock cycle. For a particular ordering of instruction types, one instruction of each type may be issued in a single clock cycle. The ordering required is A, L, E, S, B; in other words, a register-to-register address calculation, a memory load, a register-to-register data calculation, a memory store, and a branch. Because of the organization of the pipeline, each of these instructions may be serially dependent. Instructions of type E include the fixed-point execute-phase instructions as well as floating-point and digital signal processing instructions. We call this form of pipeline organization "super-string,"[4] because of the ability to issue a string of dependent instructions in a single clock cycle, as distinguished from super-scalar or super-pipelined organizations, which can only issue sets of independent instructions.

These instructions take from two to five cycles of latency to execute, and a branch prediction mechanism is used to keep the pipeline filled. The diagram below shows a box for the interval between issue of each instruction and the completion.

[4] Readers with a background in theoretical physics may have seen this term in an other, unrelated, context.

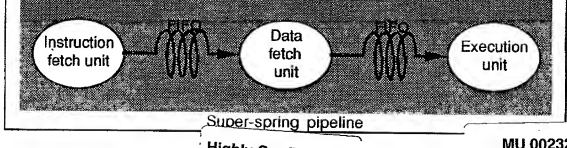
Bold letters mark the critical latency paths of the instructions, that is, the periods between the required availability of the source registers and the earliest availability of the result registers. The A-L critical latency path is a special case, in which the result of the A instruction may be used as the base register of the L instruction without penalty. E instructions may require additional cycles of latency for certain operations, such as fixed-point multiply and divide, floating-point and digital signal processing operations.



Super-spring Pipeline

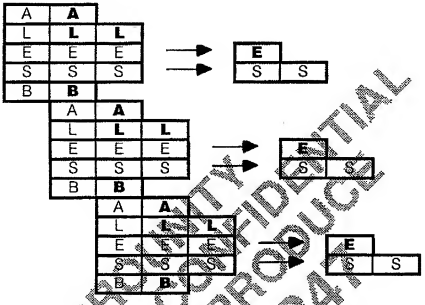
Terpsichore provides an additional refinement to the organization defined above, in which the time permitted by the pipeline to service load operations may be flexibly extended. Thus, the front of the pipeline, in which A, L and B type instructions are handled, is decoupled from the back of the pipeline, in which E, and S type instructions are handled. This decoupling occurs at the point at which the data cache and its backing memory are referenced; similarly, a FIFO that is filled by the instruction fetch unit decouples instruction cache references from the front of the pipeline shown above. The depth of the FIFO structures is implementation-dependent, i.e. not fixed by the architecture.

The diagram below indicates why we call this pipeline organization feature "super-spring," an extension of our super-string organization.



Super-spring pipeline

With the super-spring organization, the latency of load instructions can be extended, so execute instructions are deferred until the results of the load are available. Nevertheless, the execution unit still processes instructions in normal order, and provides precise exceptions.



Branch/fetch Prediction

Terpsichore does not have delayed branch instructions, and so relies upon branch or fetch prediction to keep the pipeline full around unconditional and conditional branch instructions. The hardware prediction mechanism is tuned for optimizing conditional branches that close loops or express frequent alternatives, and will generally require substantially more cycles when executing conditional branches whose outcome is not predominately taken or not-taken. For such cases, the use of code which avoids conditional branches in favor of the use of set on compare and multiplex instructions may result in greater performance.

Additional Load and Execute Resources

Studies of the dynamic distribution of Terpsichore instructions on the various benchmark suites indicate that the most frequently-issued instruction classes are load instructions and execute instructions. In a high-performance Terpsichore implementation, it is advantageous to consider execution pipelines in which the ability to target the machine resources toward issuing load and execute instructions is increased.

One of the means to increase the ability to issue execute-class instructions is to provide the means to issue two execute instructions in a single-issue string. The execution unit actually requires several distinct resources, so by partitioning these resources, the issue capability can be increased without increasing the number of functional units, other than the increased register file read and write ports. The partitioning favored for the initial implementation places all instructions that

involve shifting, including dealing, and shuffling in one execution unit, and all instructions that involve multiplication, including fixed-point and floating-point multiply and add in another unit. Resources used for implementing add, subtract, and bitwise logical operations may be duplicated, being modest in size compared to the shift and multiply units, or shared between the two units, as the operations have low-enough latency that two operations might be pipelined within a single issue cycle. These instructions must generally be independent, except perhaps that two simple add, subtract, or bitwise logical may be performed dependently, if the resources for executing simple instructions are shared between the execution units.

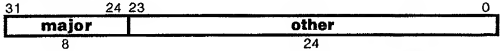
One of the means to increase the ability to issue load-class instructions is to provide the means to issue two load instructions in a single-issue string. This would generally increase the resources required of the data-fetch unit and the data cache, but a compensating solution is to steal the resources for the store instruction to execute the second load instruction. Thus, a single-issue string can then contain either two load instructions, or one load instruction and one store instruction, which uses the same register read ports and address computation resources as the basic 5-instruction string.

Result Forwarding

When temporally adjacent instructions are executed by seperate resources, the results of the first instruction must generally be forwarded directly to the resource used to execute the second instruction, where the result replaces a value which may have been fetched from a register file. Such forwarding paths use significant resources. A Terpsichore implementation must generally provide forwarding resources so that dependencies from earlier instructions within a string are immediately forwarded to later instructions, except between a first and second execution instruction as described above. In addition, when forwarding results from the execution units back to the data-fetch unit, additional delay may be incurred.

Instruction Set

All instructions are 32 bits in size, and use the high order 8 bits to specify a major operation code.

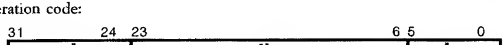


The major field is filled with a value specified by the following table:

MAJOR	0	32	64	96	128	160	192	224
0	ARES	ESETIE	FMULADD16	GMULADD1	LU16LAI	SAAS64LAI	BPE16	BE
1		ESETINE	FMULADD32	GMULADD2	LU16BAI	SAAS64BAI	BFNE16	BNE
2		ESETIL	FMULADD64	GMULADD4	LU16LI	SCAS64LAI	BPUE16	BL
3		ESETIGE	FMULADD128	GMULADD8	LU16BI	SCAS64BAI	BFNUE16	BGE
4	AADDI	EADDI	FMULSUB16	GMULSUBD16	LU32LAI	SMAS64LAI	BFNUGE16	
5		EADDIEO	FMULSUB32	GMULSUBD32	LU32BAI	SMAS64BAI	BFUGE16	
6		ESETIUL	FMULSUB64	GMULASD64	LU32LI	SMUX64LAI	BFUL16	BUL
7		ESETIUGE	FMULSUB128		LU32BI	SMUX64BAI	BFNUL16	BUGE
8		ESUBIE			L16LAI	S16BLAI	BPE32	BANDE
9		ESUBINE		GUMULACD2	L16BAI	S16BAI	BFNE32	BANDNE
10		ESUBIL		GUMULACD4	L16LI	S16LI	BPUE32	BANDL
11		ESUBIGE		GUMULACD8	L16BI	S16BI	BFNUE32	BANDGE
12	ASUBI	ESUBIAV	F.16	GUMULADD16	L32LAI	S32LAI	BFNUGE32	
13		ESUBIGAVG	F.32	GUMULASD32	L32BAI	S32BAI	BFUGE32	
14		ESUBIUL	F.64	GUMULASD64	L32LI	S32LI	BFUL32	BANDG
15		ESUBIUGE	F.128		L32BI	S32BI	BFNUL32	BANDLE
16	AANDI	EANDI	GFMULADD16	GEXTRACT128	L64LAI	S64LAI	BPE64	
17	AORI	EORI	GFMULADD32	GEXTRACT16	L64BAI	S64BAI	BFNE64	
18	AXORI	EXORI	GFMULADD64	GEXTRACT32	L64LI	S64LI	BPUE64	
19		EMUX		GMUX	L64BI	S64BI	BFNUE64	
20	ANANDI	ENANDI	GFMULSUB16	GEXTRACT64	L128LAI	S128LAI	BFNUGE64	
21	ANORI	ENORI	GFMULSUB32	GEXTRACT64	L128BAI	S128BAI	BFUGE64	BGATEI
22		EADDIUS	GFMULSUB64	GEXTRACT	L128LI	S128LI	BFUL64	
23		ESUBIUS			L128BI	S128BI	BFNUL64	
24			F.16	G.1	LUI	SII	BPE128	
25			F.32	G.2	LUBI		BFNE128	
26			F.64	G.4			BPUE128	
27			F.128	G.8			BFNUE128	
28	ACOPYI	ECOPYI	GF.16	G.16	BGATEI		BFNUGE128	BI
29			GF.32	G.32			BFUGE128	BLINKI
30			GF.64	G.64			BFUL128	
31	A.MINOR	E.MINOR		G.128	L.MINOR	S.MINOR	BFNUL128	B.MINOR

major operation code field values

For the major operation field values A.MINOR, L.MINOR, E.MINOR, F.16, F.32, F.64, F.128, GF.16, GF.32, GF.64, G.1, G.2, G.4, G.8, G.16, G.32, G.64, S.MINOR and B.MINOR, the lowest-order six bits in the instruction specify a minor operation code:



[5]Blank table entries cause the Reserved Instruction exception to occur.

MU 0023254

The minor field is filled with a value from one of the following tables:

A.MINOR	0	8	16	24	32	40	48	56
0			AAND					
1			AOR					
2			AXOR					
3			AANDN					
4	AADD	ASUB	ANAND					ASHLI
5			ANOR					
6			AXNOR					ASHRI
7			AORN					AUSHRI

minor operation code field values for A.MINOR

E.MINOR	0	8	16	24	32	40	48	56
0	ESETE	ESUBE	EAND	ESHLO	EALMS	ESHLIO		ESHLIO
1	ESETNE	ESUBNE	EOR	ESHLUO	EASUM	EDLMS		ESHLIUO
2	ESETL	ESUBL	EXOR	EEXPAND				EEXPANDI
3	ESETGE	ESUBGE	EANDN	EUEXPAND				EUEXPANDI
4	EADD	ESUB	ENAND	ESHL		EROTL	EDMUL	ESHLI
5	EADDSO	ESUBSO	ENOR	ESHLSO	EADDUO	ESUBUO		ESHLISO
6	ESETUL	ESUBUL	EXNOR	ESHRI	EGATHER	EDMUL	EROTR	ESHRI
7	ESETUGE	ESUBUGE	EORN	EUSHRI	ESCATTER			EUSHRI

minor operation code field values for E.MINOR

F.size	0	8	16	24	32	40	48	56
0	FADD.N	FADD.S	FADD.E	FADD.C	FADD	FADD.X	FSETE	FSETE.X
1	FSUB.N	FSUB.S	FSUB.E	FSUB.C	FSUB	FSUB.X	FSETNE	FSETNE.X
2	FMUL.N	FMUL.S	FMUL.E	FMUL.C	FMUL	FMUL.X	FSETUE	FSETUE.X
3	FDIV.N	FDIV.S	FDIV.E	FDIV.C	FDIV	FDIV.X	FSETNUE	FSETNUE.X
4	F.UNARY.N	F.UNARY.S	F.UNARY.E	F.UNARY.C	F.UNARY	F.UNARY.X	FSETNUGE	FSETL.X
5							FSETUGE	FSETNL.X
6							FSETUL	FSETNGE.X
7							FSETNUL	FSETGE.X

minor operation code field values for F.size

GF.size	0	8	16	24	32	40	48	56
0	GFADD.N	GFADD.T	GFADD.F	GFADD.C	GFADD	GFADD.X	GFSETE	GFSETE.X
1	GFSUB.N	GFSUB.T	GFSUB.F	GFSUB.C	GFSUB	GFSUB.X	GFSETNE	GFSETNE.X
2	GFMUL.N	GFMUL.T	GFMUL.F	GFMUL.C	GFMUL	GFMUL.X	GFSETUE	GFSETUE.X
3	GFDIV.N	GFDIV.T	GFDIV.F	GFDIV.C	GFDIV	GFDIV.X	GFSETNUE	GFSETNUE.X
4	GF.UNARY.N	GF.UNARY.T	GF.UNARY.F	GF.UNARY.C	GF.UNARY	GF.UNARY.X	GFSETNUGE	GFSETL.X
5							GFSETUGE	GFSETNL.X
6							GFSETUL	GFSETNGE.X
7							GFSETNUL	GFSETGE.X

minor operation code field values for GF.size

G.size	0	8	16	24	32	40	48	56
0	GSETE		GAND	GSETE	GCOPY	GROTL	GMUL	
1	GSETNE		GOR	GSETNE	GSWAP		GUMUL	GCOMPRESSI
2	GSETL		GXOR	GSETL	GSET		GDIV	GEXPANDI
3	GSETGE		GANDN	GSETGE	GSWIZZLE		GUDIV	GUEXPANDI
4	GADD	GSUB	GNAND	GSHL	GSCAN	GSHUFFLE	GROTR	GSHLI
5			GNOR		GBITREV			GSHRI
6	GSETUL		GXNOR	GBFIR	GGATHER	GROTR		GSHRI
7	GSETUGE		GORN	GUSHRI	GSCATTER			GUSHRI

minor operation code field values for G.size

L.MINOR	0	8	16	24	32	40	48	56
0	LU16LA	L16LA	L64LA	L8				
1	LU16BA	L16BA	L64BA	LU8				
2	LU16L	L16L	L64L					
3	LU16B	L16B	L64B					
4	LU32LA	L32LA	L128LA					
5	LU32BA	L32BA	L128BA					
6	LU32L	L32L	L128L					
7	LU32B	L32B	L128B					

minor operation code field values for L.MINOR

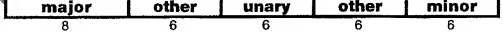
S.MINOR	0	8	16	24	32	40	48	56
0	SAAS64LA	S16LA	S64LA	S8				
1	SAAS64BA	S16BA	S64BA					
2	SCAS64LA	S16L	S64L					
3	SCAS64BA	S16B	S64B					
4	SMAS64LA	S32LA	S128LA					
5	SMAS64BA	S32BA	S128BA					
6	SMUX64LA	S32L	S128L					
7	SMUX64BA	S32B	S128B					

minor operation code field values for S.MINOR

B.MINOR	0	8	16	24	32	40	48	56
0					B			
1					BU_LINK			
2					B.DOWN			
3					B.MACC			
4				BDU128LA				
5				BDU128BA				
6				BDU128L				
7				BDU128B				

minor operation code field values for B.MINOR

For the major operation field values F.16, F.32, F.64, F.128, with minor operation field values F.UNARY.N, F.UNARY.T, F.UNARY.F, F.UNARY.C, F.UNARY, and F.UNARY.X, and for major operation field values GF.16, GF.32, GF.64, with minor operation field values GF.UNARY.N, GF.UNARY.T, GF.UNARY.F, GF.UNARY.C, GF.UNARY, and GF.UNARY.X, another six bits in the instruction specify a unary operation code:



The unary field is filled with a value from one of the following tables:

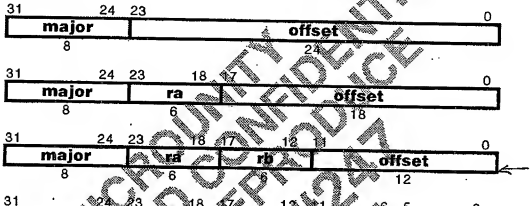
F.UNARY.size.r	0	8	16	24	32	40	48	56
0	F.ABS							
1	F.NEG							
2	F.SQR							
3								
4	F.SINK							
5	F.FLOAT							
6	F.INFLATE							
7	F.DEFLATE							

unary operation code field values for F.UNARY.size.r

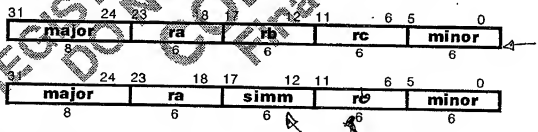
GF.UNARY.size.r	0	8	16	24	32	40	48	56
0	GF.ABS							
1	GF.NEG							
2	GF.SQR							
3								
4	GF.SINK							
5	GF.FLOAT							
6	GF.INFLATE							
7	GF.DEFLATE							

unary operation code field values for GF.UNARY.size.r

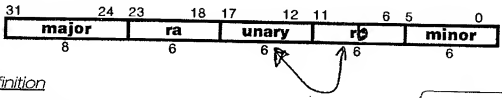
The general forms of the instructions coded by a major operation code are one of the following:



The general forms of the instructions coded by major and minor operation codes are one of the following:



The general form of the instructions coded by major, minor, and unary operation codes is the following:



Definition

```
def Instruction(inst) as
    major ← inst31..24
    ra ← inst23..18
```

```
simm ← rb ← inst17..12
rc ← inst11..6
minor ← rd ← inst5..0
case major of
    A.MINOR:
        case minor of
            A.RES
                AlwaysReserved
            A.ADD, A.AND, A.ANDN, A.NAND, A.NOR,
            A.OR, A.ORN, A.SUB, A.XNOR, A.XOR:
                Address(minor,ra,rb,rc)
            A.SHL.I, A.SHR.I, A.USHR.I:
                AddressShortImmediate(minor,ra,simm,rc)
            others:
                raise ReservedInstruction
        endcase
    A.ADD.I, A.AND.I, A.OR.I, A.NAND.I, A.NOR.I, A.XOR.I:
        AddressImmediate(major,ra,rb,inst11..0)
    A.COPY.I:
        AddressCopyImmediate(major,ra,inst17..0)
    E.MINOR:
        case minor of
            E.ADD, E.ADD.SO, E.AND, E.ANDN, E.NAND, E.NOR,
            E.OR, E.ORN, E.SUB, E.SUB.SO, E.XNOR, E.XOR,
            E.SHL, E.SHL.SO, E.SHR, E.USHR, E.EXPAND, E.UEXPAND,
            E.MUL, E.UMUL, E.DIV, E.UDIV, E.ALMS, E.ASUM,
            E.SET.E, E.SET.NE, E.SET.L, E.SET.GE, E.SET.UL, E.SET.UGE,
            E.SUB.E, E.SUB.NE, E.SUB.L, E.SUB.GE, E.SUB.UL, E.SUB.UGE:
                Execute(minor,ra,rb,rc)
            E.SHL.I, E.SHL.I.SO, E.SHR.I, E.USHR.I, E.EXPAND.I, E.UEXPAND.I:
                ExecuteShortImmediate(minor,ra,simm,rc)
            others:
                raise ReservedInstruction
        endcase
    E.ADD.I, E.ADD.I.SO, E.AND.I, E.OR.I, E.NAND.I, E.NOR.I, E.XOR.I,
    E.SET.I.E, E.SET.I.NE, E.SET.I.L, E.SET.I.GE, E.SET.I.UL, E.SET.I.UGE,
    E.SUB.I.E, E.SUB.I.NE, E.SUB.I.L, E.SUB.I.GE, E.SUB.I.UL, E.SUB.I.UGE:
        ExecuteImmediate(major,ra,rb,inst11..0)
    EMUX:
        ExecuteTernary(major,ra,rb,rc,rd)
    E.COPY.I
        ExecuteCopyImmediate(major,ra,inst17..0)
    FMULADD16, FMULADD32, FMULADD64, FMULADD128,
    FMULSUB16, FMULSUB32, FMULSUB64, FMULSUB128:
        FloatingPointTernary(major,ra,rb,rc,rd)
    F.16, F.32, F.64, F.128:
        case minor of
            F.ADD.N, F.SUB.N, F.MUL.N, F.DIV.N,
            F.ADD.T, F.SUB.T, F.MUL.T, F.DIV.T,
            F.ADD.F, F.SUB.F, F.MUL.F, F.DIV.F,
            F.ADD.C, F.SUB.C, F.MUL.C, F.DIV.C,
            F.ADD, F.SUB, F.MUL, F.DIV,
            F.ADD.X, F.SUB.X, F.MUL.X, F.DIV.X,
            F.SET.E, F.SET.NE, F.SET.UE, F.SET.NUE,
            F.SET.NUGE, F.SET.UGE, F.SET.UL, F.SET.NUL,
            F.SET.E.X, F.SET.NE.X, F.SET.UE.X, F.SET.NUE.X,
            F.SET.L.X, F.SET.NL.X, F.SET.NGE.X, F.SET.GE.X:
```

```
                FloatingPoint(minor.op, major.size, minor.round, ra, rb, rc)
            F.UNARY.N, F.UNARY.T, F.UNARY.F, F.UNARY.C,
            F.UNARY, F.UNARY.X:
                case unary of
                    F.ABS, F.NEG, F.SQR,
                    F.HALF, F.SINGLE, F.DOUBLE, F.QUAD,
                    F.INT, F.FLOAT:
                        FloatingPointUnary(unary.op, major.size, minor.round,
                                ra, rc)
                    others:
                        raise ReservedInstruction
                endcase
            others:
                raise ReservedInstruction
        endcase
    GMULADD1, GMULADD2, GMULADD4,
    GMULADD8, GMULADD16, GMULADD32,
    GUMULADD2, GUMULADD4,
    GUMULADD8, GUMULADD16, GUMULADD32,
    GMUX, GMUXGATHER, GSCATTERMUX, G.EXTRACT.128:
        GroupTernary(major.size,ra,rb,rc,rd)
    G.EXTRACT.I, G.EXTRACT.I64:
        GroupExtractImmediate(major,ra,rb,rc,minor)
    G.1, G.2, G.4, G.8, G.16, G.32:
        case minor of
            G.SHL, G.SHR, G.USHR, G.ADD, G.SUB, G.MUL, G.UMUL,
            G.AND, G.OR, G.XOR, G.ANDN, G.NAND, G.NOR, G.XNOR, G.ORN,
            G.SET.E, G.SET.NE, G.SET.L, G.SET.GE, G.SET.UL, G.SET.UGE,
            G.COPY, G.SWAP, G.DEAL, G.SHUFFLE, G.COMPRESS, G.EXPAND,
            G.GATHER, G.SCATTER:
                Group(minor,major,ra,rb,rc)
            G.COMPRESS.I, G.EXPAND.I, G.SHL.I, G.SHR.I, G.U.SHR.I:
                GroupShortImmediate(minor,major,ra,simm,rc)
            G.EXTRACT.I:
                GroupExtractImmediate(major,ra,rb,rc,minor)
            others:
                raise ReservedInstruction
        endcase
    GFMULADD16, GFMULADD32, GFMULADD64,
    GFMULSUB16, GFMULSUB32, GFMULSUB64:
        GroupFloatingPointTernary(major,ra,rb,rc,rd)
    GF.16, GF.32, GF.64, GF.128:
        case minor of
            GF.ADD.N, GF.SUB.N, GF.MUL.N, GF.DIV.N,
            GF.ADD.T, GF.SUB.T, GF.MUL.T, GF.DIV.T,
            GF.ADD.F, GF.SUB.F, GF.MUL.F, GF.DIV.F,
            GF.ADD.C, GF.SUB.C, GF.MUL.C, GF.DIV.C,
            GF.ADD, GF.SUB, GF.MUL, GF.DIV,
            GF.ADD.X, GF.SUB.X, GF.MUL.X, GF.DIV.X,
            GF.SET.E, GF.SET.NE, GF.SET.UE, GF.SET.NUE,
            GF.SET.NUGE, GF.SET.UGE, GF.SET.UL, GF.SET.NUL,
            GF.SET.E.X, GF.SET.NE.X, GF.SET.UE.X, GF.SET.NUE.X,
            GF.SET.L.X, GF.SET.NL.X, GF.SET.NGE.X, GF.SET.GE.X:
                GroupFloatingPoint(minor.op, major.size, minor.round, ra, rb, rc)
            GF.UNARY.N, GF.UNARY.T, GF.UNARY.F, GF.UNARY.C,
            GF.UNARY, GF.UNARY.X:
                case unary of
                    GF.ABS, GF.NEG, GF.SQR,
```

```
                    GF.HALF, GF.SINGLE, GF.DOUBLE, GF.QUAD,
                    GF.INT, GF.FLOAT:
                        GroupFloatingPointUnary(unary.op, major.size,
                                minor.round, ra, rc)
                    others:
                        raise ReservedInstruction
                endcase
            others:
                raise ReservedInstruction
        endcase
    L.MINOR
        case minor of
            L16L, LU16L, L32L, LU32L, L64L, L128L, L8, LU8,
            L16LA, LU16LA, L32LA, LU32LA, L64LA, L128LA,
            L16B, LU16B, L32B, LU32B, L64B, L128B,
            L16BA, LU16BA, L32BA, LU32BA, L64BA, L128BA:
                Load(minor,ra,rb,rc)
            others:
                raise ReservedInstruction
        endcase
    L16LI, LU16LI, L32LI, LU32LI, L64LI, L128LI, L8I, LU8I,
    L16LAI, LU16LAI, L32LAI, LU32LAI, L64LAI, L128LAI,
    L16BI, LU16BI, L32BI, LU32BI, L64BI, L128BI,
    L16BAI, LU16BAI, L32BAI, LU32BAI, L64BAI, L128BAI:
        LoadImmediate(major,ra,rb,inst11..0)
    S.MINOR
        case minor of
            S16L, S32L, S64L, S128L, S8,
            S16LA, S32LA, S64LA, S128LA,
            SAAS64LA, SCAS64LA, SMAS64LA, SM64LA:
            S16B, S32B, S64B, S128B,
            S16BA, S32BA, S64BA, S128BA,
            SAAS64BA, SCAS64BA, SMAS64BA, SM64BA:
                Store(minor,ra,rb,rc)
            others:
                raise ReservedInstruction
        endcase
    S16LI, S32LI, S64LI, S128LI, S8I,
    S16LAI, S32LAI, S64LAI, S128LAI,
    SAAS64LAI, SCAS64LAI, SMAS64LAI, SM64LAI,
    S16BI, S32BI, S64BI, S128BI,
    S16BAI, S32BAI, S64BAI, S128BAI
    SAAS64BAI, SCAS64BAI, SMAS64BAI, SM64BAI:
        StoreImmediate(major,ra,rb,inst11..0)
    B.MINOR:
        case minor of
            B, B.LINK, B.DOWN:
                Branch(minor,ra,rb)
            others:
                raise ReservedInstruction
        endcase
    BLINKI, BI:
        BranchImmediate(major,inst23..0)
    BFE16, BFNE16, BFUE16, BFNUE16,
    BFNUGE16, BFUGE16, BFUL16, BFNUL16,
    BFE32, BFNE32, BFUE32, BFNUE32,
    BFNUGE32, BFUGE32, BFUL32, BFNUL32,
    BFE64, BFNE64, BFUE64, BFNUE64,
```

```
        BFNUGE64, BFUGE64, BFUL64, BFNUL64,
        BFE128, BFNE128, BFUE128, BFNUE128,
        BFNUGE128, BFUGE128, BFUL128, BFNUL128,
        BE, BNE, BL, BGE, BUL, BUGE,
        BANDE, BANDNE, BANDL, BANDGE, BANDG, BANDLE:
            BranchConditional(major,inst11..0)
        BGATE:
            BranchGateway(ra,rb,inst11..0)
        others:
            raise ReservedInstruction
    endcase
enddef
```

Always Reserved

This operation generates a reserved instruction exception.

Operation code

A.RES	Always reserved

Format

A.RES imm

Description

The reserved instruction exception is raised. Software may depend upon this major operation code raising the reserved instruction exception in all Terpsichore processors. The choice of operation code intentionally ensures that a branch to a zeroed memory area will raise an exception.

Definition

```
def AlwaysReserved as
    raise ReservedInstruction
enddef
```

Exceptions

Reserved Instruction

Address

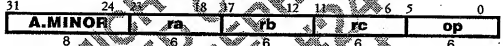
These operations perform calculations with two general register values, placing the result in a general register.

Operation codes

A.ADD	Address add
A.AND	Address and
A.ANDN	Address and not
A.NAND	Address not and
A.NOR	Address not or
A.OR	Address or
A.ORN	Address or not
A.XNOR	Address exclusive nor
A.XOR	Address xor

Format

op rc=ra,rb



Description

The contents of registers ra and rb are fetched and the specified operation is performed on these operands. The result is placed into register rc.

Definition

```
def Address(op,ra,rb,rc) as
    a ← REG[ra]
    b ← REG[rb]
    case op of
        A.ADD:
            c ← a + b
        A.AND:
            c ← a and b
        A.OR:
            c ← a or b
        A.XOR:
            c ← a xor b;
        A.ANDN:
            c ← a and not b
        A.NAND:
            c ← not (a and b)
        A.NOR:
            c ← not (a or b)
        A.XNOR:
            c ← not (a xor b)
```

```
        A.ORN:
                c ← a or not b
    endcase
    REG[rc] ← c
enddef
```

Exceptions

none

Address Copy Immediate

This operation produces one immediate value, placing the result in a general register.

Operation codes

A.COPY.I	Address copy immediate

Format

op ra=imm

Description

A 64-bit immediate value is sign-extended from the 18-bit imm field. The result is placed into register ra.

Definition

```
def AddressCopyImmediate(op,ra,imm) as
    i ← (imm17)46 || imm
    REG[ra] ← i
enddef
```

Exceptions

none

Address Immediate

These operations perform calculations with one general register value and one immediate value, placing the result in a general register.

Operation codes

A.ADD.I	Address add immediate
A.AND.I	Address and immediate
A.NAND.I	Address nand immediate
A.NOR.I	Address nor immediate
A.OR.I	Address or immediate
A.SUB.I	Address subtract immediate
A.XOR.I	Address xor immediate

Format

op rb=ra,imm

Description

The contents of register ra is fetched, and a 64-bit immediate value is sign-extended from the 12-bit imm field. The specified operation is performed on these operands. The result is placed into register rb.

Definition

```
def AddressImmediate(op,ra,rb,imm) as
    i ← (imm.52 || imm)
    a ← REG[ra]
    case op of
        A.AND.I:
            b ← a and i
        A.OR.I:
            b ← a or i
        A.NAND.I:
            b ← a nand i
        A.NOR.I:
            b ← a nor i
        A.XOR.I:
            b ← a xor i:
        A.ADD.I:
            b ← a + i
    endcase
    REG[rb] ← b
enddef
```

Exceptions

none

Address Immediate Reversed

These operations perform calculations with one general register value and one immediate value, placing the result in a general register.

Operation codes

A.SUB.I	Address subtract immediate

Format

op rb=imm,ra

Description

The contents of register ra is fetched and a 64-bit immediate value is sign-extended from the 12-bit imm field. The specified operation is performed on these operands. The result is placed into register rb.

Definition

```
def AddressImmediateReversed(op,ra,rb,imm) as
    i ← (imm12)52 || (imm)
    a ← REG[ra]
    case op of
        A.SUB.I:
            b ← i - a
    endcase
    REG[rb] ← b
enddef
```

Exceptions

none

Address Reversed

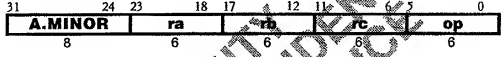
These operations perform calculations with two general register values, placing the result in a general register.

Operation codes

A.SUB	Address subtract

Format

op rc=rb,ra



Description

The contents of registers ra and rb are fetched and the specified operation is performed on these operands. The result is placed into register rc.

Definition

```
def AddressReversed(op,ra,rb,rc) as
    a ← REG[ra]
    b ← REG[rb]
    case op of
        A.SUB:
            c ← b - a
    endcase
    REG[rc] ← c
enddef
```

Exceptions

none

Address Short Immediate

These operations perform calculations with one general register value and one immediate value, placing the result in a general register.

Operation codes

A.SHL.I	Address shift left immediate
A.SHR.I	Address signed shift right immediate
A.USHR.I	Address unsigned shift right immediate

Format

op rc=ra,simm

Description

The contents of register ra is fetched, and a 6-bit immediate value is taken from the 6-bit simm field. The specified operation is performed on these operands. The result is placed into register rc.

Definition

```
def AddressShortImmediate(op,ra,simm,rc) as
    a ← REG[ra]
    case op of
        A.SHL.I:
            c ← a 63-simm..0 || 0simm
        A.SHR.I:
            c ← (a63)simm || a 63..simm
        A.USHR.I:
            c ← 0simm || a 63..simm
    endcase
    REG[rc] ← c
enddef
```

Exceptions

none

Branch

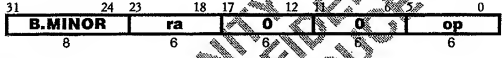
This operation branches to a location specified by a register, optionally reducing the current privilege level.

Operation codes

B	Branch
B.DOWN	Branch down in privilege

Format

op rb,ra



Description

If specified, the address of the instruction following this one is placed into register rb. Execution branches to the address specified by the contents of register ra. If specified, the current privilege level is reduced to the level specified by the low order two bits of the contents of register ra.

Access disallowed exception occurs if the contents of register ra is not aligned on a quadlet boundary, unless the operation specifies the use of the low-order two bits of the contents of register ra as a privilege level.

Definition

```
def Branch(op,ra) as
    if op ≠ B.DOWN then
        if PrivilegeLevel > REG[ra]1..0 then
            PrivilegeLevel ← REG[ra]1..0
        endif
    else
        if (REG[ra] and 3) ≠ 0 then
            raise AccessDisallowedByVirtualAddress
        endif
    endif
    ProgramCounter ← REG[ra]63..2 || 0²
enddef
```

Exceptions

Access disallowed by virtual address

Branch and Link

This operation branches to a location specified by a register, saving the value of the program counter into a register.

Operation codes

B	Branch
B.DOWN	Branch down in privilege
B.LINK	Branch and link

Format

op rb,ra

Description

If specified, the address of the instruction following this one is placed into register rb. Execution branches to the address specified by the contents of register ra. If specified, the current privilege level is reduced to the level specified by the low order two bits of the contents of register ra.

Access disallowed exception occurs if the contents of register ra is not aligned on a quadlet boundary, unless the operation specifies the use of the low-order two bits of the contents of register ra as a privilege level.

Definition

```
def BranchAndLink(op,ra,rb) as
    if (REG[ra] and 3) ≠ 0 then
        raise AccessDisallowedByVirtualAddress
    endif
    REG[rb] ← ProgramCounter + 4
    ProgramCounter ← REG[ra]63..2 || 02
enddef
```

Exceptions

Access disallowed by virtual address

Branch Conditionally

These operations compare two operands, and depending on the result of that comparison, conditionally branches to a nearby code location.

Operation codes

B.AND.E	Branch and equal to zero
B.AND.G	Branch and signed greater than zero
B.AND.GE	Branch and signed greater or equal to zero
B.AND.L	Branch and signed less than zero
B.AND.LE	Branch and signed less or equal to zero
B.AND.NE	Branch and not equal to zero
B.E[6]	Branch equal
B.F.E.16	Branch floating-point equal half
B.F.E.32	Branch floating-point equal single
B.F.E.64	Branch floating-point equal double
B.F.E.128	Branch floating-point equal quad
B.F.NE.16	Branch floating-point not equal half
B.F.NE.32	Branch floating-point not equal single
B.F.NE.64	Branch floating-point not equal double
B.F.NE.128	Branch floating-point not equal quad
B.F.NUE.16	Branch floating-point not unordered or equal half
B.F.NUE.32	Branch floating-point not unordered or equal single
B.F.NUE.64	Branch floating-point not unordered or equal double
B.F.NUE.128	Branch floating-point not unordered or equal quad
B.F.NUGE.16	Branch floating-point not unordered greater or equal half
B.F.NUGE.32	Branch floating-point not unordered greater or equal single
B.F.NUGE.64	Branch floating-point not unordered greater or equal double
B.F.NUGE.128	Branch floating-point not unordered greater or equal quad
B.F.NUL.16	Branch floating-point not unordered or less half
B.F.NUL.32	Branch floating-point not unordered or less single
B.F.NUL.64	Branch floating-point not unordered or less double
B.F.NUL.128	Branch floating-point not unordered or less quad
B.F.UE.16	Branch floating-point unordered or equal half
B.F.UE.32	Branch floating-point unordered or equal single
B.F.UE.64	Branch floating-point unordered or equal double
B.F.UE.128	Branch floating-point unordered or equal quad
B.F.UGE.16	Branch floating-point unordered greater or equal half
B.F.UGE.32	Branch floating-point unordered greater or equal single
B.F.UGE.64	Branch floating-point unordered greater or equal double
B.F.UGE.128	Branch floating-point unordered greater or equal quad
B.F.UL.16	Branch floating-point unordered or less half
B.F.UL.32	Branch floating-point unordered or less single

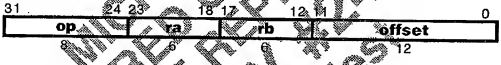
[6]B.E suffices for both signed and unsigned comparison for equality.

B.F.UL.64	Branch floating-point unordered or less double
B.F.UL.128	Branch floating-point unordered or less quad
B.GE	Branch signed greater or equal
B.L	Branch signed less
B.NE[7]	Branch not equal
B.U.GE	Branch unsigned greater or equal
B.U.L	Branch unsigned less

number format	type	compare				size
signed integer		E	NE	L	GE	
unsigned integer	U	E[8]	NE[9]	L	GE	
bitwise and	AND	E	NE	G	GE LE	
floating-point	F	E UL	NE UGE	UE NUGE	NUE NUL	16 32 64 128

Format

op rb,ra,target



Description

The contents of registers ra and rb are compared as specified by the op field. If the result of the comparison is true, execution branches to the address specified by the offset field. Otherwise, execution continues at the next sequential instruction.

If the size specified by the op field is 128, the even-odd pairs of registers specified by ra and rb are compared. In such a case, ra_0 and rb_0 must be zero for the instruction to be valid.

Definition

```
def BranchConditional(op,ra,rb,offset) as
    case op of
        BFE16, BFNE16, BFUE16, BFNUE16,
        BFNUGE16, BFUGE16, BFUL16, BFNUL16,
        BFE32, BFNE32, BFUE32, BFNUE32,
        BFNUGE32, BFUGE32, BFUL32, BFNUL32,
        BFE64, BFNE64, BFUE64, BFNUE64,
```

[7]B.NE suffices for both signed and unsigned comparison for inequality.
[8]B.U.E implemented as B.E.
[9]B.U.NE implemented as B.NE.

```
        BFNUGE64, BFUGE64, BFUL64, BFNUL64,
        BFE128, BFNE128, BFUE128, BFNUE128,
        BFNUGE128, BFUGE128, BFUL128, BFNUL128:
            type ← F
        BE, BNE, BL, BGE:
            type ← NONE
        BUL, BUGE:
            type ← U
        BANDE, BANDNE, BANDL, BANDGE, BANDG, BANDLE:
            type ← AND
    endcase
    case op of
        B.AND.G:
            compare ← G
        B.AND.LE:
            compare ← LE
        B.AND.E, B.E, B.F.E.16, B.F.E.32, B.F.E.64, B.F.E.128:
            compare ← E
        B.AND.GE, B.GE, B.U.GE:
            compare ← GE
        B.AND.L, B.L, B.U.L:
            compare ← L
        B.AND.NE, B.NE, B.F.NE.16, B.F.NE.32, B.F.NE.64, B.F.NE.128:
            compare ← NE
        B.F.UE.16, B.F.UE.32, B.F.UE.64, B.F.UE.128:
            compare ← UE
        B.F.NUE.16, B.F.NUE.32, B.F.NUE.64, B.F.NUE.128:
            compare ← NUE
        B.F.NUGE.16, B.F.NUGE.32, B.F.NUGE.64, B.F.NUGE.128:
            compare ← NUGE
        B.F.NUL.16, B.F.NUL.32, B.F.NUL.64, B.F.NUL.128:
            compare ← NUL
        B.F.UGE.16, B.F.UGE.32, B.F.UGE.64, B.F.UGE.128:
            compare ← UGE
        B.F.UL.16, B.F.UL.32, B.F.UL.64, B.F.UL.128:
            compare ← UL
    endcase
    case op of
        BFE16, BFNE16, BFUE16, BFNUE16,
        BFNUGE16, BFUGE16, BFUL16, BFNUL16:
            size ← 16
        BFE32, BFNE32, BFUE32, BFNUE32,
        BFNUGE32, BFUGE32, BFUL32, BFNUL32:
            size ← 32
        BFE64, BFNE64, BFUE64, BFNUE64,
        BFNUGE64, BFUGE64, BFUL64, BFNUL64:
            size ← 64
        BFE128, BFNE128, BFUE128, BFNUE128,
        BFNUGE128, BFUGE128, BFUL128, BFNUL128:
            size ← 128
        BE, BNE, BL, BGE, BUL, BUGE:,
        BANDE, BANDNE, BANDL, BANDGE, BANDG, BANDLE:
            size ← undefined
    endcase
    case type of
        NONE:
            l ← REG[rb]
            r ← REG[ra]
```

```
        U:
            l ← 0 || REG[rb]
            r ← 0 || REG[ra]
        AND:
            l ← REG[ra] and REG[rb]
            r ← 0
        F:
            case size of
                16
                    r ← F16(REG[ra])
                    l ← F16(REG[rb])
                32:
                    r ← F32(REG[ra])
                    l ← F32(REG[rb])
                64:
                    r ← F64(REG[ra])
                    l ← F64(REG[rb])
                128:
                    if ra0 or rb0 then
                        raise ReservedInstruction
                    else
                        l ← F128(REG[rb] || REG[rb+1])
                        r ← F128(REG[ra] || REG[ra+1])
                    endif
            endcase
    endcase
    if (type=F) and (isNaN(r) or isNaN(l)) then
        case compare of
            E, NUGE, NUL, NUE:
                c ← false
            UO, UGE, NE, UE:
                c ← true
        endcase
    else
        case compare of
            E, UE:
                c ← l = r
            NE, NUE:
                c ← l ≠ r
            L, NUGE, UL:
                c ← l < r
            NUL, UGE, GE:
                c ← l ≥ r
            G:
                c ← l > r
            LE:
                c ← l ≤ r
        endcase
    endif
    if c then
        PC ← PC + (offset11^50 || offset || 0^2)
    endif
enddef
```

Exceptions

Reserved Instruction

Branch Gateway Immediate

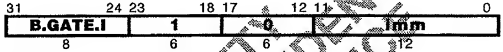
This operation provides a secure means to call a procedure, including those at a higher privilege level.

Operation codes

B.GATE.I	Branch gateway immediate

Format

B.GATE.I ra,imm



Description

A virtual address is computed from the sum of the contents of register 1 and the sign-extended value of the offset field. The contents of 16 bytes of memory using the big-endian byte order is fetched. A branch and link occurs to the contents of the memory data, and the successor to the current program counter, catenated with the current execution privilege is placed in register 0.

An access disallowed exception occurs if the target virtual address is a higher privilege than the current level and gateway access is not set for the gateway virtual address, or if the access is not aligned on a 16-byte boundary.

A reserved instruction exception occurs if the ra field is not equal to one, or the rb field is not equal to zero, or the immediate field bits 3..0 and bits 11..6 are not equal to zero.

Definition

```
def BranchGatewayImmediate(ra,rb,imm) as
    VirtAddr = REG[ra] + (imm11^50 || imm)
    if VirtAddr3..0 ≠ 0 then
        raise AccessDisallowedByVirtualAddress
    endif
    if (ra ≠ 1) or (rb ≠ 0) or (imm11..6 ≠ 0) or (imm3..0 ≠0) then
        raise ReservedInstruction
    endif
    b ← LoadMemory(VirtAddr,128,B)
    bx ← 0^64 || ProgramCounter63..2+1 || PrivilegeLevel
    ProgramCounter ← b63..2 || 0^2
    PrivilegeLevel ← LoadProtection(VirtAddr,128,B)
    REG[rb] ← bx
enddef
```

Exceptions

Reserved Instruction
Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TLB
Access disallowed by local TLB
Access detail required by tag
Access detail required by local TLB
Access detail required by global TLB
Cache coherence intervention required by tag
Cache coherence intervention required by local TLB
Cache coherence intervention required by global TLB
Local TLB miss
Global TLB miss

Branch Immediate

This operation branches to a location that is specified as an offset from the program counter, optionally saving the value of the program counter into register 0.

Operation codes

B.I	Branch immediate
B.LINK.I	Branch immediate and link

Format

op target

Description

If requested, the address of the instruction following this one is placed into register 0. Execution branches to the address specified by the offset field.

Definition

```
def BranchImmediate(op,offset) as
    if (op = B.LINK.I) then
        REG[0] ← PC + 4
    endif
    PC ← PC + (offset[23]^38 : offset : '00')
enddef
```

Exceptions

none

Execute

These operations perform calculations with two general register values, placing the result in a general register.

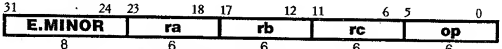
Operation codes

E.ADD	Execute add
E.ADD.SO	Execute add and check signed overflow
E.AND	Execute and
E.ANDN	Execute and not
E.ALMS	Execute and logarithm of most significant bit
E.ASUM	Execute and summation of bits
E.NAND	Execute not and
E.NOR	Execute not or
E.OR	Execute or
E.ORN	Execute or not
E.SHL	Execute shift left
E.SHL.SO	Execute shift left and check signed overflow
E.SHR	Execute signed shift right
E.U.SHR	Execute unsigned shift right
E.XNOR	Execute exclusive nor
E.XOR	Execute xor
E.GATHER	Execute gather
E.SCATTER	Execute scatter
E.EXPAND	Execute signed expand
E.U.EXPAND	Execute unsigned expand

class	operation				check
arithmetic	ADD				NONE SO
shift	SHL				NONE SO
	SHR		USHR		
	GATHER		SCATTER		
	EXPAND		U.EXPAND		
logarithm	ALMS				
summation	ASUM				
bitwise	OR	AND	XOR	ANDN	
	NOR	NAND	XNOR	ORN	

Format

op rc=ra,rb



Description

The contents of registers ra and rb are fetched and the specified operation is performed on these operands. The result is placed into register rc.

Definition

```
def Execute(op,ra,rb,rc) as
    a ← REG[ra]
    b ← REG[rb]
    case op of
        E.EXPAND:
            c ← a^32_31 || a_31..0 || 0^b4..0
        E.U.EXPAND:
            c ← 0^32_31 || a_31..0 || 0^b4..0
        E.SHL:
            c ← a 63-b5..0..0 || 0^b5..0
        E.SHL.SO:
            if a_63..63-b5..0 ≠ a_63^b5..0-1 then
                raise FixedPointArithmetic
            endif
            c ← a 63-b5..0..0 || 0^b5..0
        E.SHR:
            c ← a_63^b5..0 || a_63..b5..0
        E.USHR:
            c ← 0^b5..0 || a_63..b5..0
        E.ADD:
            c ← a + b
        E.ADD.SO:
            t ← (a_63 || a) + (b_63 || b)
            if t_64 ≠ t_63 then
                raise FixedPointArithmetic
            endif
            c ← t_63..0
        E.AND:
            c ← a and b
        E.OR:
            c ← a or b
        E.XOR:
            c ← a xor b:
        E.ANDN:
            c ← a and not b
        E.NAND:
            c ← not (a and b)
        E.NOR:
            c ← not (a or b)
        E.XNOR:
            c ← not (a xor b)
        E.ORN:
            c ← a or not b
        E.ALMS:
            t ← a & b
            if (t=0) then
```

```
                c ← -1
            else
                for i ← 0 to 63
                    if t63..i = 0^(63-i) || 1 then
                        c ← i
                    endif
                endfor
            endif
        E.ASUM:
            t ← a & b
            u ← (t63..1&0x5555555555555555) + (t&0x5555555555555555)
            v ← (u63..2&0x3333333333333333) + (u&0x3333333333333333)
            w ← (v63..4&0x70707070707070707) + (v&0x0707070707070707)
            x ← (w63..8&0xf000f000f000f) + (w&0x000f000f000f000f)
            c ← x52..48 + x36..32 + x20..16 + (x&0x1f)
        E.GATHER:
            j ← 0
            for i ← 0 to 63 by 1
                if ai then
                    cj ← bi
                    j ← j + 1
                endif
            endfor
            j ← 63
            for i ← 63 to 0 by -1
                if ~ai then
                    ci ← bj
                    j ← j - 1
                endif
            endfor
        E.SCATTER:
            j ← 0
            for i ← 0 to 63 by 1
                if ai then
                    ci ← bj
                    j ← j + 1
                endif
            endfor
            j ← 63
            for i ← 63 to 0 by -1
                if ~ai then
                    ci ← bj
                    j ← j - 1
                endif
            endfor
    endcase
    REG[rc] ← c
enddef
```

Exceptions

Fixed-point arithmetic

MU 0023281

Highly Confidential

Execute Copy Immediate

This operation produces one immediate value, placing the result in a general register.

Operation codes

E.COPY.I	Execute copy immediate

Format

E.COPY.I ra=imm

Description

A 64-bit immediate value is sign-extended from the 18-bit imm field. The result is placed into register ra.

Definition

```
def ExecuteCopyImmediate(op,ra,imm) as
    i ← (imm17^46 || imm)
    REG[ra] ← i
enddef
```

Exceptions

none

Execute Immediate

These operations perform calculations with one general register value and one immediate value, placing the result in a general register.

Operation codes

E.ADD.I	Execute add immediate
E.ADD.I.SO	Execute add immediate and check signed overflow
E.AND.I	Execute and immediate
E.NAND.I	Execute nand immediate
E.NOR.I	Execute nor immediate
E.OR.I	Execute or immediate
E.XOR.I	Execute xor immediate

class	operation	check		
arithmetic	ADD	NONE	SO	
	SUB	NONE	SO	
		E	L	UL
		NE	GE	UGE
bitwise	AND OR NAND NOR XOR			
boolean	SET.E SET.L SET.UL SET.NE SET.GE SET.UGE			

Format

op rb=ra,imm

Description

The contents of register ra is fetched, and a 64-bit immediate value is sign-extended from the 12-bit imm field. The specified operation is performed on these operands. The result is placed into register rc.

Definition

```
def ExecuteImmediate(op,ra,rb,imm) as
    i ← (imm11^52 || imm)
    a ← REG[ra]
    case op of
        E.AND.I:
            b ← a and i
        E.OR.I:
            b ← a or i
        E.NAND.I:
            b ← a nand i
```

```
        E.NOR.I:
            b ← a nor i
        E.XOR.I:
            b ← a xor i:
        E.ADD.I:
            b ← a + i
        E.ADD.I.SO:
            t ← (a63 || a) + (i63 || i)
            if t64 ≠ t63 then
                raise FixedPointArithmetic
            endif
            b ← t63..0
    endcase
    REG[rb] ← b
enddef
```

Exceptions

Fixed-point arithmetic

Execute Immediate Reversed

These operations perform calculations with one general register value and one immediate value, placing the result in a general register.

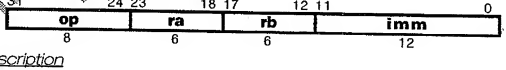
Operation codes

E.SET.I.E	Execute set immediate equal
E.SET.I.GE	Execute set immediate signed greater or equal
E.SET.I.L	Execute set immediate signed less
E.SET.I.NE	Execute set immediate not equal
E.SET.I.UGE	Execute set immediate unsigned greater or equal
E.SET.I.UL	Execute set immediate unsigned less
E.SUB.I	Execute subtract immediate and check signed less
E.SUB.I.E	Execute subtract immediate and check equal
E.SUB.I.GE	Execute subtract immediate and check signed greater or equal
E.SUB.I.L	Execute subtract immediate and check signed less
E.SUB.I.NE	Execute subtract immediate and check not equal
E.SUB.I.SO	Execute subtract immediate and check signed overflow
E.SUB.I.UGE	Execute subtract immediate and check unsigned greater or equal
E.SUB.I.UL	Execute subtract immediate and check unsigned less

class	operation	.check		
arithmetic	SUB	NONE	SO	
		E	L	UL
		NE	GE	UGE
boolean	SET.E SET.L SET.UL SET.NE SET.GE SET.UGE	*		

Format

op rb=imm,ra



Description

The contents of register ra is fetched, and a 64-bit immediate value is sign-extended from the 12-bit imm field. The specified operation is performed on these operands. The result is placed into register rc.

Definition

```
def ExecuteImmediate(op,ra,rb,imm) as
    i ← (imm11^52 || imm)
    a ← REG[ra]
    case op of
        E.SUB.I:
```

```
            b ← i - a
        E.SUB.I.SO:
            t ← (I63 || i) - (a63 || a)
            if t64 ≠ t63 then
                raise FixedPointArithmetic
            endif
            b ← t63..0
        E.SET.I.E:
            b ← (i = a)64
        E.SET.I.NE:
            b ← (i ≠ a)64
        E.SET.I.L:
            b ← (i < a)64
        E.SET.I.GE:
            b ← (i ≥ a)64
        E.SET.I.UL:
            b ← ((0 || i) < (0 || a))64
        E.SET.I.UGE:
            b ← ((0 || i) ≥ (0 || a))64
        E.SUB.I.E:
            b ← i - a
            if i ≠ a then
                raise FixedPointArithmetic
            endif
        E.SUB.I.NE:
            b ← i - a
            if i = a then
                raise FixedPointArithmetic
            endif
        E.SUB.I.L:
            b ← i - a
            if i ≥ a then
                raise FixedPointArithmetic
            endif
        E.SUB.I.GE:
            b ← i - a
            if i < a then
                raise FixedPointArithmetic
            endif
        E.SUB.I.UL:
            b ← i - a
            if (0 || i) ≥ (0 || a) then
                raise FixedPointArithmetic
            endif
        E.SUB.I.UGE:
            b ← i - a
            if (0 || i) < (0 || a) then
                raise FixedPointArithmetic
            endif
    endcase
    REG[rb] ← b
enddef
```

Exceptions

Fixed-point arithmetic

Execute Reversed

These operations perform calculations with two general register values, placing the result in a general register.

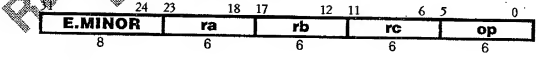
Operation codes

E.SET.E	Execute set equal
E.SET.GE	Execute set signed greater or equal
E.SET.L	Execute set signed less
E.SET.NE	Execute set not equal
E.SET.UGE	Execute set unsigned greater or equal
E.SET.UL	Execute set unsigned less
E.SUB	Execute subtract
E.SUB.E	Execute subtract and check equal
E.SUB.GE	Execute subtract and check signed greater or equal
E.SUB.L	Execute subtract and check signed less
E.SUB.NE	Execute subtract and check not equal
E.SUB.SO	Execute subtract and check signed overflow
E.SUB.UGE	Execute subtract and check unsigned greater or equal
E.SUB.UL	Execute subtract and check unsigned less

class	operation	check		
arithmetic	SUB	NONE	SO	
		E	L	UL
		NE	GE	UGE
boolean	SET.E SET.L SET.UL SET.NE SET.GE SET.UGE			

Format

op rc=rb,ra



Description

The contents of registers ra and rb are fetched and the specified operation is performed on these operands. The result is placed into register rc.

Definition

```
def ExecuteReversed(op,ra,rb,rc) as
    b ← REG[rb]
    a ← REG[ra]
    case op of
        E.SUB:
```

```
            c ← b - a
        E.SUB.SO:
            t ← (b63 || b) - (a63 || a)
            if t64 ≠ t63 then
                raise FixedPointArithmetic
            endif
            c ← t63..0
        E.SUB.E:
            c ← b - a
            if b ≠ a then
                raise FixedPointArithmetic
            endif
        E.SUB.NE:
            c ← b - a
            if b = a then
                raise FixedPointArithmetic
            endif
        E.SUB.L:
            c ← b - a
            if b ≥ a then
                raise FixedPointArithmetic
            endif
        E.SUB.GE:
            c ← b - a
            if b < a then
                raise FixedPointArithmetic
            endif
        E.SUB.UL:
            c ← b - a
            if (0 || b) ≥ (0 || a) then
                raise FixedPointArithmetic
            endif
        E.SUB.UGE:
            c ← b - a
            if (0 || b) < (0 || a) then
                raise FixedPointArithmetic
            endif
        E.SET.E:
            c ← (b = a)64
        E.SET.NE:
            c ← (b ≠ a)64
        E.SET.L:
            c ← (b < a)64
        E.SET.GE:
            c ← (b ≥ a)64
        E.SET.UL:
            c ← ((0 || b) < (0 || a))64
        E.SET.UGE:
            c ← ((0 || b) ≥ (0 || a))64
    endcase
    REG[rc] ← c
enddef
```

Exceptions

Fixed-point arithmetic

Execute Short Immediate

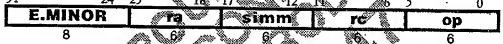
These operations perform calculations with one general register value and one immediate value, placing the result in a general register.

Operation codes

E.SHL.I	Execute shift left immediate
E.SHL.I.SO	Execute shift left immediate and check signed overflow
E.SHR.I	Execute signed shift right immediate
E.U.SHR.I	Execute unsigned shift right immediate
E.EXPAND.I	Execute signed expand immediate
E.U.EXPAND.I	Execute unsigned expand immediate

Format

op rc=ra,simm



Description

The contents of register ra is fetched, and a 6-bit immediate value is taken from the 6-bit simm field. The specified operation is performed on these operands. The result is placed into register rc.

Definition

```
def ExecuteShortImmediate(op,ra,simm,rc) as
    a ← REG[ra]
    case op of
        E.EXPAND.I:
            if simm5 then
                raise ReservedInstruction
            endif
            c ← a31^(32-simm) || a31..0 || 0^simm
        E.U.EXPAND.I:
            if simm5 then
                raise ReservedInstruction
            endif
            c ← 0^(32-simm) || a31..0 || 0^simm
        E.SHL.I:
            c ← a63-simm..0 || 0^simm
        E.SHL.I.SO:
            if a63..63-simm ≠ a63^(simm+1) then
                raise FixedPointArithmetic
            endif
            c ← a63-simm..0 || 0^simm
```

```
        E.SHR.I:
            c ← a63^simm || a 63..simm
        E.USHR.I:
            c ← 0^simm || a 63..simm
    endcase
    REG[rc] ← c
enddef
```

Exceptions

Fixed-point arithmetic

Execute Ternary

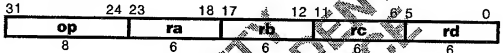
These operations perform calculations with three general register values, placing the result in a fourth general register.

Operation codes

E.MUX	Execute multiplex

Format

E.MUX rd=ra,rb,rc



Description

The contents of registers ra, rb, and rc are fetched. The specified operation is performed on these operands. The result is placed into register rd.

Definition

```
def ExecuteTernary(op,ra,rb,rc,rd) as
    a ← REG[ra]
    b ← REG[rb]
    c ← REG[rc]
    case op of
        E.MUX:
            d ← (b and a) or (c and not a)
    endcase
    REG[rd] ← d
enddef
```

Exceptions

Floating-point

These operations perform floating-point arithmetic on two floating-point operands.

Operation codes

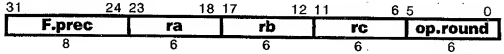
F.ADD.16	Floating-point add half
F.ADD.16.C	Floating-point add half ceiling
F.ADD.16.F	Floating-point add half floor
F.ADD.16.N	Floating-point add half nearest
F.ADD.16.T	Floating-point add half truncate
F.ADD.16.X	Floating-point add half exact
F.ADD.32	Floating-point add single
F.ADD.32.C	Floating-point add single ceiling
F.ADD.32.F	Floating-point add single floor
F.ADD.32.N	Floating-point add single nearest
F.ADD.32.T	Floating-point add single truncate
F.ADD.32.X	Floating-point add single exact
F.ADD.64	Floating-point add double
F.ADD.64 .C	Floating-point add double ceiling
F.ADD.64 .F	Floating-point add double floor
F.ADD.64 .N	Floating-point add double nearest
F.ADD.64 .T	Floating-point add double truncate
F.ADD.64 .X	Floating-point add double exact
F.ADD.128	Floating-point add quad
F.ADD.128.C	Floating-point add quad ceiling
F.ADD.128.F	Floating-point add quad floor
F.ADD.128.N	Floating-point add quad nearest
F.ADD.128.T	Floating-point add quad truncate
F.ADD.128.X	Floating-point add quad exact
F.DIV.16	Floating-point divide half
F.DIV.16.C	Floating-point divide half ceiling
F.DIV.16.F	Floating-point divide half floor
F.DIV.16.N	Floating-point divide half nearest
F.DIV.16.T	Floating-point divide half truncate
F.DIV.16.X	Floating-point divide half exact
F.DIV.32	Floating-point divide single
F.DIV.32.C	Floating-point divide single ceiling
F.DIV.32.F	Floating-point divide single floor
F.DIV.32.N	Floating-point divide single nearest
F.DIV.32.T	Floating-point divide single truncate
F.DIV.32.X	Floating-point divide single exact
F.DIV.64	Floating-point divide double
F.DIV.64.C	Floating-point divide double ceiling
F.DIV.64.F	Floating-point divide double floor
F.DIV.64.N	Floating-point divide double nearest

F.DIV.64.T	Floating-point divide double truncate
F.DIV.64.X	Floating-point divide double exact
F.DIV.128	Floating-point divide quad
F.DIV.128.C	Floating-point divide quad ceiling
F.DIV.128.F	Floating-point divide quad floor
F.DIV.128.N	Floating-point divide quad nearest
F.DIV.128.T	Floating-point divide quad truncate
F.DIV.128.X	Floating-point divide quad exact
F.MUL.16	Floating-point multiply half
F.MUL.16.C	Floating-point multiply half ceiling
F.MUL.16.F	Floating-point multiply half floor
F.MUL.16.N	Floating-point multiply half nearest
F.MUL.16.T	Floating-point multiply half truncate
F.MUL.16.X	Floating-point multiply half exact
F.MUL.32	Floating-point multiply single
F.MUL.32.C	Floating-point multiply single ceiling
F.MUL.32.F	Floating-point multiply single floor
F.MUL.32.N	Floating-point multiply single nearest
F.MUL.32.T	Floating-point multiply single truncate
F.MUL.32.X	Floating-point multiply single exact
F.MUL.64	Floating-point multiply double
F.MUL.64.C	Floating-point multiply double ceiling
F.MUL.64.F	Floating-point multiply double floor
F.MUL.64.N	Floating-point multiply double nearest
F.MUL.64.T	Floating-point multiply double truncate
F.MUL.64.X	Floating-point multiply double exact
F.MUL.128	Floating-point multiply quad
F.MUL.128.C	Floating-point multiply quad ceiling
F.MUL.128.F	Floating-point multiply quad floor
F.MUL.128.N	Floating-point multiply quad nearest
F.MUL.128.T	Floating-point multiply quad truncate
F.MUL.128.X	Floating-point multiply quad exact

	op	prec				round/trap
add	ADD	16	32	64	128	NONE C F N T X
multiply	MUL	16	32	64	128	NONE C F N T X
divide	DIV	16	32	64	128	NONE C F N T X

Format MU 0023293

F.op.prec.round rc=ra,rb



Highly Confidential

Description

The contents of registers ra and rb are combined using the specified floating-point operation. The result is placed in register rc. The operation is rounded using the specified rounding option or using round-to-nearest if not specified. If a rounding option is specified, the operation raises a floating-point exception if a floating-point invalid operation, divide by zero, overflow, or underflow occurs, or when specified, if the result is inexact. If a rounding option is not specified, floating-point exceptions are not raised, and are handled according to the default rules of IEEE 754.

Definition

```
def FloatingPoint(op,prec,round,ra,rb,rc) as
    case prec of
        16:
            a ← F16(REG[ra])
            b ← F16(REG[rb])
        32:
            a ← F32(REG[ra])
            b ← F32(REG[rb])
        64:
            a ← F64(REG[ra])
            b ← F64(REG[rb])
        128:
            a ← F128(REG[ra]
            b ← F128(REG[rb]
    endcase
    if round≠NONE then
        if isSignallingNaN(a) | isSignallingNaN(b)
            raise FloatingPointException
        endif
        case op of
            F.DIV:
                if b=0 then
                    raise FloatingPointArithmetic
                endif
            others:
        endcase
    endif
    case op of
        F.ADD:
            c ← a+b
        F.MUL:
            c ← a*b
        F.DIV.:
            c ← a/b
    endcase
    case round of
        X:
        N:
        T:
        F:
        C:
        NONE:
    endcase
    case prec of
```

```
        16:
            REG[rc] ← PackF16(c)
        32:
            REG[rc] ← PackF32(c)
        64:
            REG[rc] ← PackF64(c)
        128:
            REG[rc] ← PackF128(c)
    endcase
enddef
```

Exceptions

Reserved instruction
Floating-point arithmetic

Floating-point Reversed

These operations perform floating-point arithmetic on two floating-point operands.

Operation codes

F.SET.E.16	Floating-point set equal half
F.SET.E.16.X	Floating-point set equal half exact
F.SET.E.32	Floating-point set equal single
F.SET.E.32.X	Floating-point set equal single exact
F.SET.E.64	Floating-point set equal double
F.SET.E.64.X	Floating-point set equal double exact
F.SET.E.128	Floating-point set equal quad
F.SET.E.128.X	Floating-point set equal quad exact
F.SET.GE.16.X	Floating-point set greater or equal half exact
F.SET.GE.32.X	Floating-point set greater or equal single exact
F.SET.GE.64.X	Floating-point set greater or equal double exact
F.SET.GE.128.X	Floating-point set greater or equal quad exact
F.SET.L.16.X	Floating-point set less half exact
F.SET.L.32.X	Floating-point set less single exact
F.SET.L.64.X	Floating-point set less double exact
F.SET.L.128.X	Floating-point set less quad exact
F.SET.NE.16	Floating-point set not equal half
F.SET.NE.16.X	Floating-point set not equal half exact
F.SET.NE.32	Floating-point set not equal single
F.SET.NE.32.X	Floating-point set not equal single exact
F.SET.NE.64	Floating-point set not equal double
F.SET.NE.64.X	Floating-point set not equal double exact
F.SET.NE.128	Floating-point set not equal quad
F.SET.NE.128.X	Floating-point set not equal quad exact
F.SET.NGE.16.X	Floating-point set not greater or equal half exact
F.SET.NGE.32.X	Floating-point set not greater or equal single exact
F.SET.NGE.64.X	Floating-point set not greater or equal double exact
F.SET.NGE.128.X	Floating-point set not greater or equal quad exact
F.SET.NL.16.X	Floating-point set not or less half exact
F.SET.NL.32.X	Floating-point set not or less single exact
F.SET.NL.64.X	Floating-point set not or less double exact
F.SET.NL.128.X	Floating-point set not or less quad exact
F.SET.NUE.16	Floating-point set not unordered or equal half
F.SET.NUE.16.X	Floating-point set not unordered or equal half exact
F.SET.NUE.32	Floating-point set not unordered or equal single
F.SET.NUE.32.X	Floating-point set not unordered or equal single exact
F.SET.NUE.64	Floating-point set not unordered or equal double
F.SET.NUE.64.X	Floating-point set not unordered or equal double exact
F.SET.NUE.128	Floating-point set not unordered or equal quad
F.SET.NUE.128.X	Floating-point set not unordered or equal quad exact

F.SET.NUGE.16	Floating-point set not unordered greater or equal half
F.SET.NUGE.32	Floating-point set not unordered greater or equal single
F.SET.NUGE.64	Floating-point set not unordered greater or equal double
F.SET.NUGE.128	Floating-point set not unordered greater or equal quad
F.SET.NUL.16	Floating-point set not unordered or less half
F.SET.NUL.32	Floating-point set not unordered or less single
F.SET.NUL.64	Floating-point set not unordered or less double
F.SET.NUL.128	Floating-point set not unordered or less quad
F.SET.UE.16	Floating-point set greater or equal half
F.SET.UE.16.X	Floating-point set greater or equal half exact
F.SET.UE.32	Floating-point set greater or equal single
F.SET.UE.32.X	Floating-point set greater or equal single exact
F.SET.UE.64	Floating-point set greater or equal double
F.SET.UE.64.X	Floating-point set greater or equal double exact
F.SET.UE.128	Floating-point set greater or equal quad
F.SET.UE.128.X	Floating-point set greater or equal quad exact
F.SET.UGE.16	Floating-point set unordered greater or equal half
F.SET.UGE.32	Floating-point set unordered greater or equal single
F.SET.UGE.64	Floating-point set unordered greater or equal double
F.SET.UGE.128	Floating-point set unordered greater or equal quad
F.SET.UL.16	Floating-point set unordered or less half
F.SET.UL.32	Floating-point set unordered or less single
F.SET.UL.64	Floating-point set unordered or less double
F.SET.UL.128	Floating-point set unordered or less quad
F.SUB.16	Floating-point subtract half
F.SUB.16.C	Floating-point subtract half ceiling
F.SUB.16.F	Floating-point subtract half floor
F.SUB.16.N	Floating-point subtract half nearest
F.SUB.16.T	Floating-point subtract half truncate
F.SUB.16.X	Floating-point subtract half exact
F.SUB.32	Floating-point subtract single
F.SUB.32.C	Floating-point subtract single ceiling
F.SUB.32.F	Floating-point subtract single floor
F.SUB.32.N	Floating-point subtract single nearest
F.SUB.32.T	Floating-point subtract single truncate
F.SUB.32.X	Floating-point subtract single exact
F.SUB.64	Floating-point subtract double
F.SUB.64.C	Floating-point subtract double ceiling
F.SUB.64.F	Floating-point subtract double floor
F.SUB.64.N	Floating-point subtract double nearest
F.SUB.64.T	Floating-point subtract double truncate
F.SUB.64.X	Floating-point subtract double exact
F.SUB.128	Floating-point subtract quad
F.SUB.128.C	Floating-point subtract quad ceiling
F.SUB.128.F	Floating-point subtract quad floor
F.SUB.128.N	Floating-point subtract quad nearest

MU 0023297

F.SUB.128.T	Floating-point subtract quad truncate
F.SUB.128.X	Floating-point subtract quad exact

	op		prec				round/trap
set	SET. E UE	 NE NUE	16	32	64	128	NONE X
	SET. NUGE UGE	 NUL UL	16	32	64	128	NONE
	SET. L NL	 GE NGE	16	32	64	128	X
subtract	SUB		16	32	64	128	NONE I F N T X

Format

F.op.prec.round rc=rb,ra

Description

The contents of registers ra and rb are combined using the specified floating-point operation. The result is placed in register rc. The operation is rounded using the specified rounding option or using round-to-nearest if not specified. If a rounding option is specified, the operation raises a floating-point exception if a floating-point invalid operation, divide by zero, overflow, or underflow occurs, or when specified, if the result is inexact. If a rounding option is not specified, floating-point exceptions are not raised, and are handled according to the default rules of IEEE 754.

Definition

```
def FloatingPointReversed(op,prec,round,ra,rb,rc) as
    case prec of
        16:
            a ← F16(REG[ra])
            b ← F16(REG[rb])
        32:
            a ← F32(REG[ra])
            b ← F32(REG[rb])
        64:
            a ← F64(REG[ra])
            b ← F64(REG[rb])
        128:
            a ← F128(REG[ra]
            b ← F128(REG[rb]
    endcase
    if round≠NONE then
```

```
        if isSignallingNaN(a) | isSignallingNaN(b)
            raise FloatingPointException
        endif
        case op of
            F.SET.L, F.SET.GE, F.SET.NL, F.SET.NGE:
                if isNaN(a) | isNaN(b) then
                    raise FloatingPointArithmetic
                endif
            others:
        endcase
    endif
    case op of
        F.SUB:
            c ← b-a
        F.SET.NUGE, F.SET.L:
            c ← b!?≥a
        F.SET.NUL, F.SET.GE:
            c ← b!?<a
        F.SET.UGE, F.SET.NL:
            c ← b?≥a
        F.SET.UL, F.SET.NGE:
            c ← b?<a
        F.SET.UE:
            c ← b?=a
        F.SET.NUE:
            c ← b!?=a
        F.SET.E:
            c ← b=a
        F.SET.NE:
            c ← b≠a
    endcase
    case op of
        F.SUB:
            destprec ← prec
        F.SET.NUGE, F.SET.NUL, F.SET.UGE, F.SET.UL,
        F.SET.L, F.SET.GE, F.SET.E, F.SET.NE, F.SET.UE, F.SET.NUE:
            destprec ← INT
    endcase
    case round of
        X:
        N:
        T:
        F:
        C:
        NONE:
    endcase
    case destprec of
        16:
            REG[rc] ← PackF16(c)
        32:
            REG[rc] ← PackF32(c)
        64:
            REG[rc] ← PackF64(c)
        128:
            REG[rc] ← PackF128(c)
        INT:
            REG[rc] ← c
```

```
    endcase
enddef
```

Exceptions

Reserved instruction
Floating-point arithmetic

Floating-point Ternary

These operations perform floating-point arithmetic on three floating-point operands..

Operation codes

F.MULADD.16	Floating-point multiply and add half
F.MULADD.32	Floating-point multiply and add single
F.MULADD.64	Floating-point multiply and add double
F.MULADD.128	Floating-point multiply and add quad
F.MULSUB.16	Floating-point multiply and subtract half
F.MULSUB.32	Floating-point multiply and subtract single
F.MULSUB.64	Floating-point multiply and subtract double
F.MULSUB.128	Floating-point multiply and subtract quad

	op	prec			
multiply and add	MULADD	16	32	64	128
multiply and subtract	MULSUB	16	32	64	128

Format

F.operation.type rd=ra,rb,rc

Description

The contents of registers ra and rb are multiplied together and added to or subtracted from the contents of register rc. The result is placed in register rd. The result is rounded to the nearest representable floating-point value in a single floating-point operation. Floating-point exceptions are not raised, and are handled according to the default rules of IEEE 754. These instructions cannot select a directed rounding mode or trap on inexact.

Definition

```
def FloatingPointTernary(op,ra,rb,rc,rd) as
    case op of
        FMULADD16, FMULSUB16:
            a ← F16(REG[ra])
            b ← F16(REG[rb])
            c ← F16(REG[rc])
        FMULADD32, FMULSUB32:
            a ← F32(REG[ra])
            b ← F32(REG[rb])
            c ← F32(REG[rc])
        FMULADD64, FMULSUB64:
```

```
            a ← F64(REG[ra])
            b ← F64(REG[rb])
            c ← F64(REG[rc])
        FMULADD128, FMULSUB128:
            a ← F128(REG[ra])
            b ← F128(REG[rb])
            c ← F128(REG[rc])
    endcase
    case op of
        FMULADD16, FMULADD32, FMULADD64, FMULADD128:
            d ← a*b+c
        FMULSUB16, FMULSUB32, FMULSUB64, FMULSUB128:
            d ← a*b-c
    endcase
    case op of
        FMULADD16, FMULSUB16:
            REG[rd] ← PackF16(d)
        FMULADD32, FMULSUB32:
            REG[rd] ← PackF32(d)
        FMULADD64, FMULSUB64:
            REG[rd] ← PackF64(d)
        FMULADD128, FMULSUB128:
            REG[rd] ← PackF128(d)
    endcase
enddef
```

Exceptions

Reserved instruction
Floating-point arithmetic

Floating-point Unary

These operations perform floating-point arithmetic on one floating-point operand.

Operation codes

F.ABS.16	Floating-point absolute value half
F.ABS.16.X	Floating-point absolute value half exact
F.ABS.32	Floating-point absolute value single
F.ABS.32.X	Floating-point absolute value single exact
F.ABS.64	Floating-point absolute value double
F.ABS.64.X	Floating-point absolute value double exact
F.ABS.128	Floating-point absolute value quad
F.ABS.128.X	Floating-point absolute value quad exact
F.DEFLATE.32	Floating-point convert half from single
F.DEFLATE.32.C	Floating-point convert half from single ceiling
F.DEFLATE.32.F	Floating-point convert half from single floor
F.DEFLATE.32.N	Floating-point convert half from single nearest
F.DEFLATE.32.T	Floating-point convert half from single truncate
F.DEFLATE.32.X	Floating-point convert half from single exact
F.DEFLATE.64	Floating-point convert single from double
F.DEFLATE.64.C	Floating-point convert single from double ceiling
F.DEFLATE.64.F	Floating-point convert single from double floor
F.DEFLATE.64.N	Floating-point convert single from double nearest
F.DEFLATE.64.T	Floating-point convert single from double truncate
F.DEFLATE.64.X	Floating-point convert single from double exact
F.DEFLATE.128	Floating-point convert double from quad
F.DEFLATE.128.C	Floating-point convert double from quad ceiling
F.DEFLATE.128.F	Floating-point convert double from quad floor
F.DEFLATE.128.N	Floating-point convert double from quad nearest
F.DEFLATE.128.T	Floating-point convert double from quad truncate
F.DEFLATE.128.X	Floating-point convert double from quad exact
F.FLOAT.16	Floating-point convert half from integer
F.FLOAT.16.C	Floating-point convert half from integer ceiling
F.FLOAT.16.F	Floating-point convert half from integer floor
F.FLOAT.16.N	Floating-point convert half from integer nearest
F.FLOAT.16.T	Floating-point convert half from integer truncate
F.FLOAT.16.X	Floating-point convert half from integer exact
F.FLOAT.32	Floating-point convert single from integer
F.FLOAT.32.C	Floating-point convert single from integer ceiling
F.FLOAT.32.F	Floating-point convert single from integer floor
F.FLOAT.32.N	Floating-point convert single from integer nearest
F.FLOAT.32.T	Floating-point convert single from integer truncate
F.FLOAT.32.X	Floating-point convert single from integer exact
F.FLOAT.64	Floating-point convert double from integer
F.FLOAT.64.C	Floating-point convert double from integer ceiling

MU 0023303

F.FLOAT.64.F	Floating-point convert double from integer floor
F.FLOAT.64.N	Floating-point convert double from integer nearest
F.FLOAT.64.T	Floating-point convert double from integer truncate
F.FLOAT.64.X	Floating-point convert double from integer exact
F.FLOAT.128	Floating-point convert quad from integer
F.INFLATE.16	Floating-point convert single from half
F.INFLATE.16.X	Floating-point convert single from half exact
F.INFLATE.32	Floating-point convert double from single
F.INFLATE.32.X	Floating-point convert double from single exact
F.INFLATE.64	Floating-point convert quad from double
F.INFLATE.64.X	Floating-point convert quad from double exact
F.NEG.16	Floating-point negate half
F.NEG.16.X	Floating-point negate half exact
F.NEG.32	Floating-point negate single
F.NEG.32.X	Floating-point negate single exact
F.NEG.64	Floating-point negate double
F.NEG.64.X	Floating-point negate double exact
F.NEG.128	Floating-point negate quad
F.NEG.128.X	Floating-point negate quad exact
F.SINK.16	Floating-point convert integer from half
F.SINK.16.C	Floating-point convert integer from half ceiling
F.SINK.16.F	Floating-point convert integer from half floor
F.SINK.16.N	Floating-point convert integer from half nearest
F.SINK.16.T	Floating-point convert integer from half truncate
F.SINK.16.X	Floating-point convert integer from half exact
F.SINK.32	Floating-point convert integer from single
F.SINK.32.C	Floating-point convert integer from single ceiling
F.SINK.32.F	Floating-point convert integer from single floor
F.SINK.32.N	Floating-point convert integer from single nearest
F.SINK.32.T	Floating-point convert integer from single truncate
F.SINK.32.X	Floating-point convert integer from single exact
F.SINK.64	Floating-point convert integer from double
F.SINK.64.C	Floating-point convert integer from double ceiling
F.SINK.64.F	Floating-point convert integer from double floor
F.SINK.64.N	Floating-point convert integer from double nearest
F.SINK.64.T	Floating-point convert integer from double truncate
F.SINK.64.X	Floating-point convert integer from double exact
F.SINK.128	Floating-point convert integer from quad
F.SINK.128.C	Floating-point convert integer from quad ceiling
F.SINK.128.F	Floating-point convert integer from quad floor
F.SINK.128.N	Floating-point convert integer from quad nearest
F.SINK.128.T	Floating-point convert integer from quad truncate
F.SINK.128.X	Floating-point convert integer from quad exact
F.SQR.16	Floating-point square root half
F.SQR.16.C	Floating-point square root half ceiling
F.SQR.16.F	Floating-point square root half floor

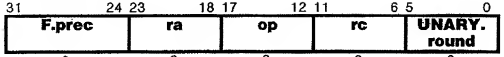
MU 0023304

F.SQR.16.N	Floating-point square root half nearest
F.SQR.16.T	Floating-point square root half truncate
F.SQR.16.X	Floating-point square root half exact
F.SQR.32	Floating-point square root single
F.SQR.32.C	Floating-point square root single ceiling
F.SQR.32.F	Floating-point square root single floor
F.SQR.32.N	Floating-point square root single nearest
F.SQR.32.T	Floating-point square root single truncate
F.SQR.32.X	Floating-point square root single exact
F.SQR.64	Floating-point square root double
F.SQR.64.C	Floating-point square root double ceiling
F.SQR.64.F	Floating-point square root double floor
F.SQR.64.N	Floating-point square root double nearest
F.SQR.64.T	Floating-point square root double truncate
F.SQR.64.X	Floating-point square root double exact
F.SQR.128	Floating-point square root quad
F.SQR.128.C	Floating-point square root quad ceiling
F.SQR.128.F	Floating-point square root quad floor
F.SQR.128.N	Floating-point square root quad nearest
F.SQR.128.T	Floating-point square root quad truncate
F.SQR.128.X	Floating-point square root quad exact

	Op	prec				round/trap
absolute value	ABS	16	32	64	128	NONE X
float from integer	FLOAT	16	32	64		NONE C F N T X
					128	NONE
integer from float	SINK	16	32	64	128	NONE C F N T X
increase format precision	INFLATE	16	32	64		NONE X
decrease format precision	DEFLATE		32	64	128	NONE C F N T X
negate	NEG	16	32	64	128	NONE X
square root	SQR	16	32	64	128	NONE C F N T X

Format

F.op.prec.round rc=ra



Description

The contents of register ra is used as the operand of the specified floating-point operation. The result is placed in register rc. The operation is rounded using the specified rounding option or using round-to-nearest if not specified. If a rounding option is specified, the operation raises a floating-point exception if a floating-point invalid operation, divide by zero, overflow, or underflow occurs, or when specified, if the result is inexact. If a rounding option is not specified, floating-point exceptions are not raised, and are handled according to the default rules of IEEE 754.

Definition

```
def FloatingPointUnary(op,prec,round,ra,rb,rc) as
    if op = F.FLOAT then
            a ← REG[ra]
    else
        case prec of
            16:
                a ← F16(REG[ra])
            32:
                a ← F32(REG[ra])
            64:
                a ← F64(REG[ra])
            128:
                a ← F128(REG[ra])
        endcase
    endif
    case op of
        F.ABS:
            if a < 0 then
                c ← -a
            else
                c ← a
            endif
        F.NEG:
            c ← -a
        F.SQR:
            c ← √a
        F.FLOAT, F.SINK, F.INFLATE, F.DEFLATE:
            c ← a
    endcase
    case op of
        F.ABS, F.NEG, F.SQR, F.FLOAT:
            destprec ← prec
        F.SINK
```


MU 0023306

```
                destprec ← INT
            F.INFLATE:
                destprec ← prec + prec
            F.DEFLATE:
                destprec ← prec / 2
        endcase
        case round of
            X:
            N:
            T:
            F:
            C:
            NONE:
        endcase
        case destprec of
            16:
                REG[rc] ← PackF16(c)
            32:
                REG[rc] ← PackF32(c)
            64:
                REG[rc] ← PackF64(c)
            128:
                REG[rc] ← PackF128(c)
            INT:
                REG[rc] ← c
        endcase
enddef
```

Exceptions

Reserved instruction
Floating-point arithmetic

Group

These operations take two values from a pair of registers, perform operations on groups of bits in the operands, and place the concatenated results in a register .

Operation codes

G.ADD.2	Group add pecks
G.ADD.4	Group add nibbles
G.ADD.8	Group add bytes
G.ADD.16	Group add doublets
G.ADD.32	Group add quadlets
G.ADD.64	Group add octlets
G.AND[10]	Group and
G.ANDN[11]	Group and not
G.COMPRESS.1	Group compress bits
G.COMPRESS.2	Group compress pecks
G.COMPRESS.4	Group compress nibbles
G.COMPRESS.8	Group compress bytes
G.COMPRESS.16	Group compress doublets
G.COMPRESS.32	Group compress quadlets
G.COMPRESS.64	Group compress octlets
G.COPY.1	Group copy bits
G.COPY.2	Group copy pecks
G.COPY.4	Group copy nibbles
G.COPY.8	Group copy bytes
G.COPY.16	Group copy doublets
G.COPY.32	Group copy quadlets
G.COPY.64	Group copy octlets
G.DEAL.1	Group deal bits
G.DEAL.2	Group deal pecks
G.DEAL.4	Group deal nibbles
G.DEAL.8	Group deal bytes
G.DEAL.16	Group deal doublets
G.DEAL.32	Group deal quadlets
G.DIV.64	Group signed divide octlets
G.EXPAND.1	Group signed expand bits
G.EXPAND.2	Group signed expand pecks
G.EXPAND.4	Group signed expand nibbles
G.EXPAND.8	Group signed expand bytes
G.EXPAND.16	Group signed expand doublets
G.EXPAND.32	Group signed expand quadlets
G.EXPAND.64	Group signed expand octlet

[10]G.AND does not require a size specification, and is encoded as G.AND.1.
[11]G.ANDN does not require a size specification, and is encoded as G.ANDN.1. G.ANDN is used as the encoding for G.SET.L.1, and by reversing the operands, for G.SET.UL.1.

G.GATHER.2	Group gather pecks
G.GATHER.4	Group gather nibbles
G.GATHER.8	Group gather bytes
G.GATHER.16	Group gather doublets
G.GATHER.32	Group gather quadlets
G.GATHER.64	Group gather octlets
G.GATHER.128[12]	Group gather hexlets
G.MUL.1[13]	Group signed multiply bits
G.MUL.2	Group signed multiply pecks
G.MUL.4	Group signed multiply nibbles
G.MUL.8	Group signed multiply bytes
G.MUL.16	Group signed multiply doublets
G.MUL.32	Group signed multiply quadlets
G.MUL.64	Group signed multiply octlets
G.NAND[14]	Group nand
G.NOR[15]	Group nor
G.OR[16]	Group or
G.ORN[17]	Group or not
G.POLY.1	Group polynomial divide bits
G.POLY.2	Group polynomial divide pecks
G.POLY.4	Group polynomial divide nibbles
G.POLY.8	Group polynomial divide bytes
G.POLY.16	Group polynomial divide doublets
G.POLY.32	Group polynomial divide quadlets
G.POLY.64	Group polynomial divide octlets
G.SCATTER.2	Group scatter pecks
G.SCATTER.4	Group scatter nibbles
G.SCATTER.8	Group scatter bytes
G.SCATTER.16	Group scatter doublets
G.SCATTER.32	Group scatter quadlets
G.SCATTER.64	Group scatter octlets
G.SCATTER.128[18]	Group scatter hexlet
G.SHL.2	Group shift left pecks
G.SHL.4	Group shift left nibbles
G.SHL.8	Group shift left bytes
G.SHL.16	Group shift left doublets
G.SHL.32	Group shift left quadlets
G.SHL.64	Group shift left octlets

[12] G.GATHER.128 is encoded as G.GATHER.1

[13] G.MUL.1 is used as the encoding for G.UMUL.1.

[14] G.NAND does not require a size specification, and is encoded as G.NAND.1.

[15] G.NOR does not require a size specification, and is encoded as G.NOR.1.

[16] G.OR does not require a size specification, and is encoded as G.OR.1.

[17] G.ORN does not require a size specification, and is encoded as G.ORN.1. G.ORN is used as the encoding for G.SET.UGE.1, and by reversing the operands, for G.SET.GE.1.

[18] G.SCATTER.128 is encoded as G.SCATTER.1

G.SHR.2	Group signed shift right pecks
G.SHR.4	Group signed shift right nibbles
G.SHR.8	Group signed shift right bytes
G.SHR.16	Group signed shift right doublets
G.SHR.32	Group signed shift right quadlets
G.SHR.64	Group signed shift right octlets
G.SHUFFLE.1	Group shuffle bits
G.SHUFFLE.2	Group shuffle pecks
G.SHUFFLE.4	Group shuffle nibbles
G.SHUFFLE.8	Group shuffle bytes
G.SHUFFLE.16	Group shuffle doublets
G.SHUFFLE.32	Group shuffle quadlets
G.SWAP.1	Group swap bits
G.SWAP.2	Group swap pecks
G.SWAP.4	Group swap nibbles
G.SWAP.8	Group swap bytes
G.SWAP.16	Group swap doublets
G.SWAP.32	Group swap quadlets
G.U.DIV.64	Group signed divide octlets
G.U.EXPAND.1	Group unsigned expand bits
G.U.EXPAND.2	Group unsigned expand pecks
G.U.EXPAND.4	Group unsigned expand nibbles
G.U.EXPAND.8	Group unsigned expand bytes
G.U.EXPAND.16	Group unsigned expand doublets
G.U.EXPAND.32	Group unsigned expand quadlets
G.U.EXPAND.64	Group unsigned expand octlet
G.U.MUL.2	Group unsigned multiply pecks
G.U.MUL.4	Group unsigned multiply nibbles
G.U.MUL.8	Group unsigned multiply bytes
G.U.MUL.16	Group unsigned multiply doublets
G.U.MUL.32	Group unsigned multiply quadlets
G.U.MUL.64	Group unsigned multiply octlets
G.U.SHR.2	Group unsigned shift right pecks
G.U.SHR.4	Group unsigned shift right nibbles
G.U.SHR.8	Group unsigned shift right bytes
G.U.SHR.16	Group unsigned shift right doublets
G.U.SHR.32	Group unsigned shift right quadlets
G.U.SHR.64	Group unsigned shift right octlets
G.XNOR[19]	Group exclusive-nor
G.XOR[20]	Group exclusive-or

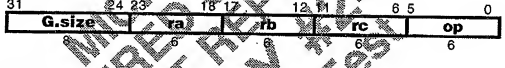
[19]G.XNOR does not require a size specification, and is encoded as G.XNOR.1. G.XNOR is used as the encoding for G.SET.E.1.

[20]G.XOR does not require a size specification, and is encoded as G.XOR.1. G.XOR is used as the encoding for G.ADD.1, G.SUB.1 and G.SET.NE.1.

class	op	size
linear	ADD	2 4 8 16 32 64
bitwise	AND ANDN NAND NOR OR ORN XNOR XOR	
signed multiply	MUL	1 2 4 8 16 32 64
unsigned multiply	U.MUL	2 4 8 16 32 64
signed divide	DIV	64
unsigned divide	U.DIV	64
rearrange	COPY DEAL SWAP SHUFFLE	1 2 4 8 16 32
	GATHER SCATTER	2 4 8 16 32 64
galois field	POLY	1 2 4 8 16 32 64
precision	COMPRESS EXPAND U.EXPAND	1 2 4 8 16 32 64
shift	SHL SHR U.SHR	2 4 8 16 32 64

Format

G.op.size rc=ra,rb



Description

Two values are taken from the contents of registers ra and rb. The specified operation is performed, and the result is placed in register rc.

Definition

```
def Group(op,size,ra,rb,rc)
    case op of
        G.MUL, G.U.MUL, G.DIV, G.U.DIV:
            a ← REG[ra]
            b ← REG[rb]
        G.ADD, G.SUB, G.SET.L, G.SET.UL, G.SET.E, G.SET.NE, G.SET.GE, G.SET.UGE,
        G.AND, G.OR, G.XOR, G.ANDN, G.NAND, G.NOR, G.XNOR, G.ORN,
        G.GATHER, G.SCATTER:
            a ← REG[ra]
            b ← REG[rb]
        G.COMPRESS, G.SHL, G.SHR, G.U.SHR, G.POLY:
            a ← REG[ra]
            b ← REG[rb]
        G.EXPAND, G.U.EXPAND:
            a ← REG[ra]
            b ← REG[rb]
        G.COPY, G.SWAP, G.DEAL, G.SHUFFLE:
            a ← REG[ra] || REG[rb]
    endcase
```

```
case op of
    G.ADD:
        for i ← 0 to 128-size by size
            c_{i+size-1..i} ← a_{i+size-1..i} + b_{i+size-1..i}
        endfor
    G.MUL:
        for i ← 0 to 64-size by size
            c_{2*(i+size)-1..2*i} ← (a_{size-1}^size || a_{size-1+i..i}) * (b_{size-1}^size || b_{size-1+i..i})
        endfor
    G.U.MUL:
        for i ← 0 to 64-size by size
            c_{2*(i+size)-1..2*i} ← (0^size || a_{size-1+i..i}) * (0^size || b_{size-1+i..i})
        endfor
    G.DIV:
        if (b = 0) or ( (a = (1||0^63)) and (b = 1^64) ) then
            c ← undefined
        else
            q ← a / b
            r ← a - q*b
            c ← r_{63..0} || q_{63..0}
        endif
    G.U.DIV:
        if b = 0 then
            c ← undefined
        else
            q ← (0.|| a) / (0 ||b)
            r ← a - q*b
            c ← r_{63..0} || q_{63..0}
        endif
    G.AND:
        c ← a and b
    G.OR:
        c ← a or b
    G.XOR:
        c ← a xor b
    G.ANDN:
        c ← a and not b
    G.NAND:
        c ← not (a and b)
    G.NOR:
        c ← not (a or b)
    G.XNOR:
        c ← not (a xor b)
    G.ORN:
        c ← a or not b
    G.POLY:
        p[0] ← a
        for i ← 1 to size
            p[i] ← (p[i-1]_0 ? (0^64 || b) : 0^128) xor (p[i-1]_0 || p[i-1]_{127..1})
        endfor
        c ← p[size]
    G.GATHER:
        for k ← 0 to 128-size by size
            j ← k
            for i ← k to k+size-1 by 1
                if a_i then
                    c_j ← b_i
```

```
                    j ← j + 1
                endif
            endfor
            j ← k+size
            for i ← k+size-1 to k by -1
                if ~ai then
                    cj ← bi
                    j ← j - 1
                endif
            endfor
        endfor
    G.SCATTER:
        for k ← 0 to 128-size by size
            j ← k
            for i ← k to k+size-1 by 1
                if ai then
                    ci ← bj
                    j ← j + 1
                endif
            endfor
            j ← k+size-1
            for i ← k+size-1 to k by -1
                if ~ai then
                    ci ← bj
                    j ← j - 1
                endif
            endfor
        endfor
    G.COMPRESS:
        for i ← 0 to 64-size by size
            ci+size-1..i ← ai+i+size-1+(b&(size-1))..i+i+(b&(size-1))
        endfor
    G.EXPAND:
        for i ← 0 to 64-size by size
            ci+i+size+size-1..i+i ← a i+size-1 || 0^(size-(b&(size-1))) || ai+size-1..i || 0^(b&(size-1))
        endfor
    G.U.EXPAND:
        for i ← 0 to 64-size by size
            ci+i+size+size-1..i+i ← 0^(size-(b&(size-1))) || ai+size-1..i || 0^(b&(size-1))
        endfor
    G.SHL:
        for i ← 0 to 128-size by size
            ci+size-1..i ← ai+size-1-(b&(size-1))..i || 0^(b&(size-1))
        endfor
    G.SHR:
        for i ← 0 to 128-size by size
            ci+size-1..i ← ai+size-1^(b&(size-1)) || ai+size-1..i+(b&(size-1))
        endfor
    G.U.SHR:
        for i ← 0 to 128-size by size
            ci+size-1..i ← 0^(b&(size-1)) || ai+size-1..i+(b&(size-1))
        endfor
    G.COPY:
        for i ← 0 to 128-size by size
            ci+size-1..i ← asize-1..0
```

```
            endfor
        G.SWAP:
            for i ← 0 to 128-size by size
                c_{i+size-1..i} ← a_{127-i..128-size-i}
            endfor
        G.DEAL:
            for i ← 0 to 128-size by size
                j ← (i_{5..0} || 0^1)+(i_6 ? size : 0)
                c_{i+size-1..i} ← a_{j+size-1..j}
            endfor
        G.SHUFFLE:
            for i ← 0 to 128-size by size
                j ← (0^1 || i_{6..1})+((i&size) ? (64-(0^1 || size_{6..1})) : 0)
                c_{i+size-1..i} ← a_{j+size-1..j}
            endfor
    endcase
        REG[rc] ← c
enddef
```

Exceptions

Reserved Instruction

Group Extract Immediate

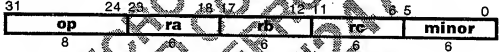
These operations perform calculations with two general register values and a small immediate field, placing the result in a third general register.

Operation codes

G.EXTRACT.I.1	Group extract immediate bits
G.EXTRACT.I.2	Group extract immediate pecks
G.EXTRACT.I.4	Group extract immediate nibbles
G.EXTRACT.I.8	Group extract immediate bytes
G.EXTRACT.I.16	Group extract immediate doublets
G.EXTRACT.I.32	Group extract immediate quadlets
G.EXTRACT.I.64	Group extract immediate octlets
G.EXTRACT.I.128	Group extract immediate hexlet

Format

G.EXTRACT.I.size rc←ra,rb,shift



Description

The contents of registers pairs specified by ra, and rb are fetched. The specified operation is performed on these operands. The result is placed into the register pair specified by rc.

Definition

```
def GroupExtractImmediate(op,ra,rb,rc,minor) as
  ab ← REG[ra] || REG[rb]
  case op of
    G.64:
        size ← 64
        shift ← 64
    G.32:
        size ← 32
        shift ← 32
    G.16:
        size ← 16
        shift ← 16
    G.8:
        size ← 8
        shift ← 8
    G.4:
        size ← 4
        shift ← 4
    G.2:
        size ← 2
        shift ← 2
```

```
        G.1:
            size ← 1
            shift ← 1
        G.EXTRACT.I:
            case minor of
                0..31:
                    size ← 32
                    shift ← minor
                32..47:
                    size ← 16
                    shift ← minor - 32
                48..55:
                    size ← 8
                    shift ← minor - 48
                56..59:
                    size ← 4
                    shift ← minor - 56
                60..61:
                    size ← 2
                    shift ← minor - 60
                62:
                    size ← 1
                    shift ← 0
                63:
                    raise ReservedInstruction
            endcase
        G.EXTRACT.I.64:
            size ← 64
            shift ← minor
        G.EXTRACT.I.128:  I+G.EXTRACT.I.128:
            size ← 128
            shift ← opc || minor
    endcase
    for i ← 0 to 128-size by size
        c[i+size-1..i] ← ab[i+shift+size-1..i+shift]
    endfor
    REG[rc] ← c
enddef
```

Exceptions

Reserved instruction

Group Reversed

These operations take two values from a pair of registers, perform operations on groups of bits in the operands, and place the concatenated results in a register.

Operation codes

G.SET.E.2	Group set equal pecks
G.SET.E.4	Group set equal nibbles
G.SET.E.8	Group set equal bytes
G.SET.E.16	Group set equal doublets
G.SET.E.32	Group set equal quadlets
G.SET.E.64	Group set equal octlets
G.SET.GE.2	Group set signed greater or equal pecks
G.SET.GE.4	Group set signed greater or equal nibbles
G.SET.GE.8	Group set signed greater or equal bytes
G.SET.GE.16	Group set signed greater or equal doublets
G.SET.GE.32	Group set signed greater or equal quadlets
G.SET.GE.64	Group set signed greater or equal octlets
G.SET.L.2	Group set signed less pecks
G.SET.L.4	Group set signed less nibbles
G.SET.L.8	Group set signed less bytes
G.SET.L.16	Group set signed less doublets
G.SET.L.32	Group set signed less quadlets
G.SET.L.64	Group set signed less octlets
G.SET.NE.2	Group set not equal pecks
G.SET.NE.4	Group set not equal nibbles
G.SET.NE.8	Group set not equal bytes
G.SET.NE.16	Group set not equal doublets
G.SET.NE.32	Group set not equal quadlets
G.SET.NE.64	Group set not equal octlets
G.SET.UGE.2	Group set unsigned greater or equal pecks
G.SET.UGE.4	Group set unsigned greater or equal nibbles
G.SET.UGE.8	Group set unsigned greater or equal bytes
G.SET.UGE.16	Group set unsigned greater or equal doublets
G.SET.UGE.32	Group set unsigned greater or equal quadlets
G.SET.UGE.64	Group set unsigned greater or equal octlets
G.SET.UL.2	Group set unsigned less pecks
G.SET.UL.4	Group set unsigned less nibbles
G.SET.UL.8	Group set unsigned less bytes
G.SET.UL.16	Group set unsigned less doublets
G.SET.UL.32	Group set unsigned less quadlets
G.SET.UL.64	Group set unsigned less octlets
G.SUB.2	Group subtract pecks
G.SUB.4	Group subtract nibbles
G.SUB.8	Group subtract bytes

G.SUB.16	Group subtract doublets
G.SUB.32	Group subtract quadlets
G.SUB.64	Group subtract octlets

class	op	size
linear	SUB	2 4 8 16 32 64
boolean	SET.E SET.L SET.GE SET.NE SET.UL SET.UGE	2 4 8 16 32 64

Format

G.op.size rc=rb,ra

Description

Two values are taken from the contents of registers ra and rb. The specified operation is performed, and the result is placed in register rc.

Definition

```
def GroupReverse(op,size,ra,rb,rc)
        a ← REG[ra]
        b ← REG[rb]
    case op of
        G.SUB:
            for i ← 0 to 128-size by size
                c_{i+size-1..i} ← b_{i+size-1..i} - a_{i+size-1..i}
            endfor
        G.SET.L:
            for i ← 0 to 128-size by size
                c_{i+size-1..i} ← (b_{i+size-1..i} < a_{i+size-1..i})^size
            endfor
        G.SET.UL:
            for i ← 0 to 128-size by size
                c_{i+size-1..i} ← (0 || b_{+size-1..i} < 0 || a_{i+size-1..i})^size
            endfor
        G.SET.E:
            for i ← 0 to 128-size by size
                c_{i+size-1..i} ← (b_{i+size-1..i} = a_{i+size-1..i})^size
            endfor
        G.SET.NE:
            for i ← 0 to 128-size by size
                c_{i+size-1..i} ← (b_{i+size-1..i} ≠ a_{i+size-1..i})^size
            endfor
        G.SET.GE:
            for i ← 0 to 128-size by size
                c_{i+size-1..i} ← (b_{i+size-1..i} ≥ a_{i+size-1..i})^size
            endfor
        G.SET.UGE:
```

```
            for i ← 0 to 128-size by size
                c_{i+size-1..i} ← (0 || b_{i+size-1..i} ≥ 0 || a_{i+size-1..i})^size
            endfor
    endcase
        REG[rc] c
enddef
```

Exceptions

Reserved Instruction

Group Short Immediate

These operations take two values from a pair of registers, perform operations on groups of bits in the operands, and place the concatenated results in a register.

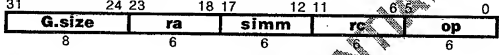
Operation codes

G.COMPRESS.I.1	Group compress immediate bits
G.COMPRESS.I.2	Group compress immediate pecks
G.COMPRESS.I.4	Group compress immediate nibbles
G.COMPRESS.I.8	Group compress immediate bytes
G.COMPRESS.I.16	Group compress immediate doublets
G.COMPRESS.I.32	Group compress immediate quadlets
G.COMPRESS.I.64	Group compress immediate octlet
G.EXPAND.I.1	Group signed expand immediate bits
G.EXPAND.I.2	Group signed expand immediate pecks
G.EXPAND.I.4	Group signed expand immediate nibbles
G.EXPAND.I.8	Group signed expand immediate bytes
G.EXPAND.I.16	Group signed expand immediate doublets
G.EXPAND.I.32	Group signed expand immediate quadlets
G.EXPAND.I.64	Group signed expand immediate octlet
G.SHL.I.2	Group shift left immediate pecks
G.SHL.I.4	Group shift left immediate nibbles
G.SHL.I.8	Group shift left immediate bytes
G.SHL.I.16	Group shift left immediate doublets
G.SHL.I.32	Group shift left immediate quadlets
G.SHL.I.64	Group shift left immediate octlets
G.SHR.I.2	Group signed shift right immediate pecks
G.SHR.I.4	Group signed shift right immediate nibbles
G.SHR.I.8	Group signed shift right immediate bytes
G.SHR.I.16	Group signed shift right immediate doublets
G.SHR.I.32	Group signed shift right immediate quadlets
G.SHR.I.64	Group signed shift right immediate octlets
G.U.EXPAND.I.1	Group unsigned expand immediate bits
G.U.EXPAND.I.2	Group unsigned expand immediate pecks
G.U.EXPAND.I.4	Group unsigned expand immediate nibbles
G.U.EXPAND.I.8	Group unsigned expand immediate bytes
G.U.EXPAND.I.16	Group unsigned expand immediate doublets
G.U.EXPAND.I.32	Group unsigned expand immediate quadlets
G.U.EXPAND.I.64	Group unsigned expand immediate octlet
G.U.SHR.I.2	Group unsigned shift right immediate pecks
G.U.SHR.I.4	Group unsigned shift right immediate nibbles
G.U.SHR.I.8	Group unsigned shift right immediate bytes
G.U.SHR.I.16	Group unsigned shift right immediate doublets
G.U.SHR.I.32	Group unsigned shift right immediate quadlets
G.U.SHR.I.64	Group unsigned shift right immediate octlets

class	op			size						
precision	COMPRESS.I		EXPAND.I U.EXPAND.I	1	2	4	8	16	32	64
shift	SHL.I	SHR.I	U.SHR.I		2	4	8	16	32	64

Format

G.op.size rc=ra,simm



Description

A 128-bit value is taken from the contents of register ra. The second operand is taken from simm. The specified operation is performed, and the result is placed in register rc.

This instruction is undefined and causes a reserved instruction exception if the simm field is greater or equal to the size specified.

Definition

```
def GroupShortImmediate(op,size,ra,simm,rc)
    a ← REG[ra]
    if simm≥size then
        raise ReservedInstruction
    endif
    case op of
        G.COMPRESS.I:
            for i ← 0 to 64-size by size
                c[i+size-1..i] ← a[i+i+size-1+simm..i+i+simm]
            endfor
        G.EXPAND.I:
            for i ← 0 to 64-size by size
                c[i+i+size+size-1..i+i] ← a[i+size-1+size-simm] || a[i+size-1..i] || 0^simm
            endfor
        G.U.EXPAND.I:
            for i ← 0 to 64-size by size
                c[i+i+size+size-1..i+i] ← 0^(size-simm) || a[i+size-1..i] || 0^simm
            endfor
        G.SHL.I:
            for i ← 0 to 128-size by size
                c[i+size-1..i] ← a[i+size-1-simm..i] || 0^simm
            endfor
        G.SHR.I:
            for i ← 0 to 128-size by size
                c[i+size-1..i] ← a[i+size-1]^simm || a[i+size-1..i+simm]
            endfor
        G.U.SHR.I:
```

```
            for i ← 0 to 128-size by size
                c[i+size-1..i] ← 0^simm ||| a[i+size-1..i+simm]
            endfor
    endcase
        REG[rc] ← c
enddef
```

Exceptions

Reserved Instruction

Group Ternary

These operations perform calculations with three general register values, placing the result in a fourth general register.

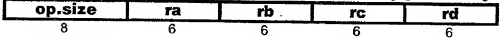
Operation codes

G.EXTRACT.128	Group extract hexlet
G.MULADD.1[21]	Group signed multiply bits and add pecks
G.MULADD.2	Group signed multiply pecks and add nibbles
G.MULADD.4	Group signed multiply nibbles and add bytes
G.MULADD.8	Group signed multiply bytes and add doublets
G.MULADD.16	Group signed multiply doublets and add quadlets
G.MULADD.32	Group signed multiply quadlets and add octlets
G.MULADD.64	Group signed multiply octlets and add hexlets
G.MUX	Group multiplex
G.MUX.GATHER	Group multiplex and gather
G.SCATTER.MUX	Group scatter and multiplex
G.U.MULADD.2	Group unsigned multiply pecks and add nibbles
G.U.MULADD.4	Group unsigned multiply nibbles and add bytes
G.U.MULADD.8	Group unsigned multiply bytes and add doublets
G.U.MULADD.16	Group unsigned multiply doublets and add quadlets
G.U.MULADD.32	Group unsigned multiply quadlets and add octlets
G.U.MULADD.64	Group unsigned multiply octlets and add hexlets

class	op	prec
extract	EXTRACT	128
signed multiply and add	MULADD	1 2 4 8 16 32 64
unsigned multiply and add	U.MULADD	2 4 8 16 32 64
multiplex	MUX MUX.GATHER SCATTER.MUX	NONE

Format

G.op.size rd=ra,rb,rc



[21] G.MULADD.1 is used as the encoding for G.UMULADD.1.

Description

The contents of registers ra, rb, and rc are fetched. The specified operation is performed on these operands. The result is placed into register rd.

Definition

```
def GroupTernary(op,size,ra,rb,rc,rd) as
        a ← REG[ra]
        b ← REG[rb]
    c ← REG[rc]
    case op of
        G.MUX:
            d ← (b and a) or (c andnot a)
        G.MUX.GATHER:
            t ← (b and a) or (c andnot a)
            j ← 0
            for i ← 0 to 127 by 1
                if ai then
                    dj ← ti
                    j ← j + 1
                endif
            endfor
            j ← 127
            for i ← 127 to 0 by -1
                if ¬ai then
                    dj ← ti
                    j ← j-1
                endif
            endfor
        G.SCATTER.MUX:
            j ← 0
            for i ← 0 to 127 by 1
                if ai then
                    di ← bj
                    j ← j + 1
                else
                    di ← ci
                endif
            endfor
        endfor
        G.EXTRACT:
            d ← (a || b)(c&127)+127..(c&127)
        G.MULADD:
            for i ← 0 to 64-size by size
                d2*(i+size)-1..2*i ← c2*(i+size)-1..2*i +
                        (a_size-1^size || a_size-1+i..i) * (b_size-1^size || b_size-1+i..i)
            endfor
        G.U.MULADD:
            for i ← 0 to 64-size by size
                d2*(i+size)-1..2*i ← c2*(i+size)-1..2*i +
                        (0^size || a_size-1+i..i) * (0^size || b_size-1+i..i)
            endfor
    endcase
    REG[rd] ← d
enddef
```

MU 0023324

Highly Confidential

Exceptions

Reserved instruction

Group Floating-point

These operations take two values from registers, perform floating-point arithmetic on groups of bits in the operands, and place the concatenated results in a register.

Operation codes

GF.ADD.16	Group floating-point add half
GF.ADD.16.C	Group floating-point add half ceiling
GF.ADD.16.F	Group floating-point add half floor
GF.ADD.16.N	Group floating-point add half nearest
GF.ADD.16.T	Group floating-point add half truncate
GF.ADD.16.X	Group floating-point add half exact
GF.ADD.32	Group floating-point add single
GF.ADD.32.C	Group floating-point add single ceiling
GF.ADD.32.F	Group floating-point add single floor
GF.ADD.32.N	Group floating-point add single nearest
GF.ADD.32.T	Group floating-point add single truncate
GF.ADD.32.X	Group floating-point add single exact
GF.ADD.64	Group floating-point add double
GF.ADD.64 .C	Group floating-point add double ceiling
GF.ADD.64 .F	Group floating-point add double floor
GF.ADD.64 .N	Group floating-point add double nearest
GF.ADD.64 .T	Group floating-point add double truncate
GF.ADD.64 .X	Group floating-point add double exact
GF.DIV.16	Group floating-point divide half
GF.DIV.16.C	Group floating-point divide half ceiling
GF.DIV.16.F	Group floating-point divide half floor
GF.DIV.16.N	Group floating-point divide half nearest
GF.DIV.16.T	Group floating-point divide half truncate
GF.DIV.16.X	Group floating-point divide half exact
GF.DIV.32	Group floating-point divide single
GF.DIV.32.C	Group floating-point divide single ceiling
GF.DIV.32.F	Group floating-point divide single floor
GF.DIV.32.N	Group floating-point divide single nearest
GF.DIV.32.T	Group floating-point divide single truncate
GF.DIV.32.X	Group floating-point divide single exact
GF.DIV.64	Group floating-point divide double
GF.DIV.64.C	Group floating-point divide double ceiling
GF.DIV.64.F	Group floating-point divide double floor
GF.DIV.64.N	Group floating-point divide double nearest
GF.DIV.64.T	Group floating-point divide double truncate
GF.DIV.64.X	Group floating-point divide double exact
GF.MUL.16	Group floating-point multiply half
GF.MUL.16.C	Group floating-point multiply half ceiling
GF.MUL.16.F	Group floating-point multiply half floor

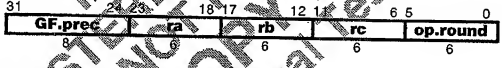
MU 0023326

GF.MUL.16.N	Group floating-point multiply half nearest
GF.MUL.16.T	Group floating-point multiply half truncate
GF.MUL.16.X	Group floating-point multiply half exact
GF.MUL.32	Group floating-point multiply single
GF.MUL.32.C	Group floating-point multiply single ceiling
GF.MUL.32.F	Group floating-point multiply single floor
GF.MUL.32.N	Group floating-point multiply single nearest
GF.MUL.32.T	Group floating-point multiply single truncate
GF.MUL.32.X	Group floating-point multiply single exact
GF.MUL.64	Group floating-point multiply double
GF.MUL.64.C	Group floating-point multiply double ceiling
GF.MUL.64.F	Group floating-point multiply double floor
GF.MUL.64.N	Group floating-point multiply double nearest
GF.MUL.64.T	Group floating-point multiply double truncate
GF.MUL.64.X	Group floating-point multiply double exact

	op	prec	round/trap
add	ADD	16 32 64	NONE C F N T X
divide	DIV	16 32 64	NONE C F N T X
multiply	MUL	16 32 64	NONE C F N T X

Format

GF.op.prec.round rc=ra,rb



Description

The contents of registers ra and rb are combined using the specified floating-point operation. The result is placed in register rc. The operation is rounded using the specified rounding option or using round-to-nearest if not specified. If a rounding option is specified, the operation raises a floating-point exception if a floating-point invalid operation, divide by zero, overflow, or underflow occurs, or when specified, if the result is inexact. If a rounding option is not specified, floating-point exceptions are not raised, and are handled according to the default rules of IEEE 754.

Definition

```
def GroupFloatingPoint(op,prec,round,ra,rb,rc) as
    a ← REG[ra] !
    b ← REG[rb]
    for i ← 0 to 128-prec by prec
        ai ← F(prec,ai+prec-1..i)
        bi ← F(prec,bi+prec-1..i)
        if round≠NONE then
```

```
            if isSignallingNaN(ai) | isSignallingNaN(bi)
                raise FloatingPointException
            endif
            case op of
                F.DIV:
                    if bi=0 then
                        raise FloatingPointArithmetic
                    endif
                others:
            endcase
        endif
        case op of
            GF.ADD:
                ci ← ai+bi
            GF.MUL:
                ci ← ai*bi
            GF.DIV.:
                ci ← ai/bi
        endcase
        case op of
            GF.ADD, GF.MUL, GF.DIV:
                ci+prec-1..i ← RecipF(prec,ci)
        endcase
    endfor
    endcase
    case round of
        X:
        N:
        T:
        F:
        C:
        NONE:
    endcase
    if rc0 then
        raise ReservedInstruction
    endif
    REG[rc] ← c
    endcase
enddef
```

Exceptions

Reserved instruction
Floating-point arithmetic

Group Floating-point Reversed

These operations take two values from registers, perform floating-point arithmetic on groups of bits in the operands, and place the concatenated results in a register.

Operation codes

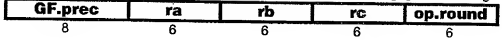
GF.SET.E.16	Group floating-point set equal half
GF.SET.E.16.X	Group floating-point set equal half exact
GF.SET.E.32	Group floating-point set equal single
GF.SET.E.32.X	Group floating-point set equal single exact
GF.SET.E.64	Group floating-point set equal double
GF.SET.E.64.X	Group floating-point set equal double exact
GF.SET.GE.16.X	Group floating-point set greater or equal half exact
GF.SET.GE.32.X	Group floating-point set greater or equal single exact
GF.SET.GE.64.X	Group floating-point set greater or equal double exact
GF.SET.L.16.X	Group floating-point set less half exact
GF.SET.L.32.X	Group floating-point set less single exact
GF.SET.L.64.X	Group floating-point set less double exact
GF.SET.NE.16	Group floating-point set not equal half
GF.SET.NE.16.X	Group floating-point set not equal half exact
GF.SET.NE.32	Group floating-point set not equal single
GF.SET.NE.32.X	Group floating-point set not equal single exact
GF.SET.NE.64	Group floating-point set not equal double
GF.SET.NE.64.X	Group floating-point set not equal double exact
GF.SET.NGE.16.X	Group floating-point set not greater or equal half exact
GF.SET.NGE.32.X	Group floating-point set not greater or equal single exact
GF.SET.NGE.64.X	Group floating-point set not greater or equal double exact
GF.SET.NL.16.X	Group floating-point set not less half exact
GF.SET.NL.32.X	Group floating-point set not less single exact
GF.SET.NL.64.X	Group floating-point set not less double exact
GF.SET.NUE.16	Group floating-point set not unordered or equal half
GF.SET.NUE.16.X	Group floating-point set not unordered or equal half exact
GF.SET.NUE.32	Group floating-point set not unordered or equal single
GF.SET.NUE.32.X	Group floating-point set not unordered or equal single exact
GF.SET.NUE.64	Group floating-point set not unordered or equal double
GF.SET.NUE.64.X	Group floating-point set not unordered or equal double exact
GF.SET.NUGE.16	Group floating-point set not unordered greater or equal half
GF.SET.NUGE.32	Group floating-point set not unordered greater or equal single
GF.SET.NUGE.64	Group floating-point set not unordered greater or equal double
GF.SET.NUL.16	Group floating-point set not unordered or less half
GF.SET.NUL.32	Group floating-point set not unordered or less single
GF.SET.NUL.64	Group floating-point set not unordered or less double
GF.SET.UE.16	Group floating-point set unordered or equal half
GF.SET.UE.16.X	Group floating-point set unordered or equal half exact
GF.SET.UE.32	Group floating-point set unordered or equal single

GF.SET.UE.32.X	Group floating-point set unordered or equal single exact
GF.SET.UE.64	Group floating-point set unordered or equal double
GF.SET.UE.64.X	Group floating-point set unordered or equal double exact
GF.SET.UGE.16	Group floating-point set unordered greater or equal half
GF.SET.UGE.32	Group floating-point set unordered greater or equal single
GF.SET.UGE.64	Group floating-point set unordered greater or equal double
GF.SET.UL.16	Group floating-point set unordered or less half
GF.SET.UL.32	Group floating-point set unordered or less single
GF.SET.UL.64	Group floating-point set unordered or less double
GF.SUB.16	Group floating-point subtract half
GF.SUB.16.C	Group floating-point subtract half ceiling
GF.SUB.16.F	Group floating-point subtract half floor
GF.SUB.16.N	Group floating-point subtract half nearest
GF.SUB.16.T	Group floating-point subtract half truncate
GF.SUB.16.X	Group floating-point subtract half exact
GF.SUB.32	Group floating-point subtract single
GF.SUB.32.C	Group floating-point subtract single ceiling
GF.SUB.32.F	Group floating-point subtract single floor
GF.SUB.32.N	Group floating-point subtract single nearest
GF.SUB.32.T	Group floating-point subtract single truncate
GF.SUB.32.X	Group floating-point subtract single exact
GF.SUB.64	Group floating-point subtract double
GF.SUB.64.C	Group floating-point subtract double ceiling
GF.SUB.64.F	Group floating-point subtract double floor
GF.SUB.64.N	Group floating-point subtract double nearest
GF.SUB.64.T	Group floating-point subtract double truncate
GF.SUB.64.X	Group floating-point subtract double exact

	op		prec	round/trap
set	SET. E UE	 NE NUE	16 32 64	NONE X
	SET. NUGE UGE	 NUL UL	16 32 64	NONE
	SET. L NL	 GE NGE	16 32 64	X
subtract	SUB		16 32 64	NONE C F N T X

Format

GF.op.prec.round rc=rb,ra



Description

The contents of registers ra and rb are combined using the specified floating-point operation. The result is placed in register rc. The operation is rounded using the specified rounding option or using round-to-nearest if not specified. If a rounding option is specified, the operation raises a floating-point exception if a floating-point invalid operation, divide by zero, overflow, or underflow occurs, or when specified, if the result is inexact. If a rounding option is not specified, floating-point exceptions are not raised, and are handled according to the default rules of IEEE 754.

Definition

```
def GroupFloatingPointReversed(op,pres,round,ra,rb,rc) as
    If ra0 or rb0 then
        raise ReservedInstruction
    endif
    a ← REG[ra]
    b ← REG[rb]
    for i ← 0 to 128-prec by prec
        ai ← F(prec,ai<prec-1:i>)
        bi ← F(prec,bi<prec-1:i>)
        if round=NoNE then
            if isSignallingNaN(ai) | isSignallingNaN(bi)
                raise FloatingPointException
            endif
            case op of
                GF.SET.L, GF.SET.GE, GF.SET.NL, GF.SET.NGE:
                    if isNaN(ai) | isNaN(bi) then
                        raise FloatingPointArithmetic
                    endif
            endcase
        endif
        case op of
            GF.SUB:
                ci ← bi-ai
            GF.SET.NUGE, GF.SET.L:
                ci ← bi?≥ai
            GF.SET.NUL, GF.SET.GE:
                ci ← bi|?<ai
            GF.SET.UGE, GF.SET.NL:
                ci ← bi?≥ai
            GF.SET.UL, GF.SET.NGE:
                ci ← bi?<ai
            GF.SET.UE:
                ci ← b?=ai
            GF.SET.NUE:
```

```
                ci ← bII?=ai
            GF.SET.E:
                ci ← bi=ai
            GF.SET.NE:
                ci ← bi≠ai
        endcase
        case op of
            GF.SUB:
                ci+prec-1..i ← PackF(prec, ci)
            GF.SET.NUGE, GF.SET.NUL, GF.SET.UGE, GF.SET.UL,
            GF.SET.L, GF.SET.GE, GF.SET.E, GF.SET.NE, GF.SET.UE, GF.SET.NUE:
                ci+prec-1..i ← ci
        endcase
    endfor
    endcase
    case round of
        X:
        N:
        T:
        F:
        C:
        NONE:
    endcase
    REG[rc] ← c
    endcase
enddef
```

Exceptions

Reserved instruction
Floating-point arithmetic

Group Floating-point Ternary

These operations perform floating-point arithmetic on three groups of floating-point operands contained in registers.

Operation codes

GF.MULADD.16	Group floating-point multiply and add half
GF.MULSUB.16	Group floating-point multiply and subtract half
GF.MULADD.32	Group floating-point multiply and add single
GF.MULSUB.32	Group floating-point multiply and subtract single
GF.MULADD.64	Group floating-point multiply and add double
GF.MULSUB.64	Group floating-point multiply and subtract double

	op	prec		
multiply and add	MULADD	16	32	64
multiply and subtract	MULSUB	16	32	64

Format

GF.operation.type rd=ra,rb,rc

Description

The contents of registers ra and rb are taken to represent a group of floating-point operands and pairwise are multiplied together and added to or subtracted from the group of floating-point operands taken from the contents of register rc. The results are concatenated and placed in register . The results are rounded to the nearest representable floating-point value in a single floating-point operation. Floating-point exceptions are not raised, and are handled according to the default rules of IEEE 754. These instructions cannot select a directed rounding mode or trap on inexact.

Definition

```
def GroupFloatingPointTernary(op,prec,ra,rb,rc,rd) as
    a ← REG[ra]
    b ← REG[rb]
    c ← REG[rc]
    for i ← 0 to 128-prec by prec
        ai ← F(prec,ai+prec-1..i)
        bi ← F(prec,bi+prec-1..i)
        ci ← F(prec,ci+prec-1..i)
        case op of
            GF.MULADD:
                di ← (ai * bi ) + ci
```

```
            GF.MULSUB:
                di ← (ai * bi ) - ci
        endcase
        di+prec-1..i ← PackF(prec, di)
    endfor
    REG[rd] ← d
enddef
```

Exceptions

Reserved instruction
Floating-point arithmetic

Group Floating-point Unary

MU 0023335

These operations take one value from a register , perform floating-point arithmetic on groups of bits in the operands, and place the concatenated results in a register .

Operation codes

GF.ABS.16	Group floating-point absolute value half
GF.ABS.16.X	Group floating-point absolute value half exact
GF.ABS.32	Group floating-point absolute value single
GF.ABS.32.X	Group floating-point absolute value single exact
GF.ABS.64	Group floating-point absolute value double
GF.ABS.64.X	Group floating-point absolute value double exact
GF.DEFLATE.32	Group floating-point convert half from single
GF.DEFLATE.32.C	Group floating-point convert half from single ceiling
GF.DEFLATE.32.F	Group floating-point convert half from single floor
GF.DEFLATE.32.N	Group floating-point convert half from single nearest
GF.DEFLATE.32.T	Group floating-point convert half from single truncate
GF.DEFLATE.32.X	Group floating-point convert half from single exact
GF.DEFLATE.64	Group floating-point convert single from double
GF.DEFLATE.64.C	Group floating-point convert single from double ceiling
GF.DEFLATE.64.F	Group floating-point convert single from double floor
GF.DEFLATE.64.N	Group floating-point convert single from double nearest
GF.DEFLATE.64.T	Group floating-point convert single from double truncate
GF.DEFLATE.64.X	Group floating-point convert single from double exact
GF.FLOAT.16	Group floating-point convert half from integer
GF.FLOAT.16.C	Group floating-point convert half from integer ceiling
GF.FLOAT.16.F	Group floating-point convert half from integer floor
GF.FLOAT.16.N	Group floating-point convert half from integer nearest
GF.FLOAT.16.T	Group floating-point convert half from integer truncate
GF.FLOAT.16.X	Group floating-point convert half from integer exact
GF.FLOAT.32	Group floating-point convert single from integer
GF.FLOAT.32.C	Group floating-point convert single from integer ceiling
GF.FLOAT.32.F	Group floating-point convert single from integer floor
GF.FLOAT.32.N	Group floating-point convert single from integer nearest
GF.FLOAT.32.T	Group floating-point convert single from integer truncate
GF.FLOAT.32.X	Group floating-point convert single from integer exact
GF.FLOAT.64	Group floating-point convert double from integer
GF.FLOAT.64.C	Group floating-point convert double from integer ceiling
GF.FLOAT.64.F	Group floating-point convert double from integer floor
GF.FLOAT.64.N	Group floating-point convert double from integer nearest
GF.FLOAT.64.T	Group floating-point convert double from integer truncate
GF.FLOAT.64.X	Group floating-point convert double from integer exact
GF.INFLATE.16	Group floating-point convert single from half
GF.INFLATE.16.X	Group floating-point convert single from half exact
GF.INFLATE.32	Group floating-point convert double from single

GF.INFLATE.32.X	Group floating-point convert double from single exact
GF.NEG.16	Group floating-point negate half
GF.NEG.16.X	Group floating-point negate half exact
GF.NEG.32	Group floating-point negate single
GF.NEG.32.X	Group floating-point negate single exact
GF.NEG.64	Group floating-point negate double
GF.NEG.64.X	Group floating-point negate double exact
GF.SINK.16	Group floating-point convert integer from half
GF.SINK.16.C	Group floating-point convert integer from half ceiling
GF.SINK.16.F	Group floating-point convert integer from half floor
GF.SINK.16.N	Group floating-point convert integer from half nearest
GF.SINK.16.T	Group floating-point convert integer from half truncate
GF.SINK.16.X	Group floating-point convert integer from half exact
GF.SINK.32	Group floating-point convert integer from single
GF.SINK.32.C	Group floating-point convert integer from single ceiling
GF.SINK.32.F	Group floating-point convert integer from single floor
GF.SINK.32.N	Group floating-point convert integer from single nearest
GF.SINK.32.T	Group floating-point convert integer from single truncate
GF.SINK.32.X	Group floating-point convert integer from single exact
GF.SINK.64	Group floating-point convert integer from double
GF.SINK.64.C	Group floating-point convert integer from double ceiling
GF.SINK.64.F	Group floating-point convert integer from double floor
GF.SINK.64.N	Group floating-point convert integer from double nearest
GF.SINK.64.T	Group floating-point convert integer from double truncate
GF.SINK.64.X	Group floating-point convert integer from double exact
GF.SQR.16	Group floating-point square root half
GF.SQR.16.C	Group floating-point square root half ceiling
GF.SQR.16.F	Group floating-point square root half floor
GF.SQR.16.N	Group floating-point square root half nearest
GF.SQR.16.T	Group floating-point square root half truncate
GF.SQR.16.X	Group floating-point square root half exact
GF.SQR.32	Group floating-point square root single
GF.SQR.32.C	Group floating-point square root single ceiling
GF.SQR.32.F	Group floating-point square root single floor
GF.SQR.32.N	Group floating-point square root single nearest
GF.SQR.32.T	Group floating-point square root single truncate
GF.SQR.32.X	Group floating-point square root single exact
GF.SQR.64	Group floating-point square root double
GF.SQR.64.C	Group floating-point square root double ceiling
GF.SQR.64.F	Group floating-point square root double floor
GF.SQR.64.N	Group floating-point square root double nearest
GF.SQR.64.T	Group floating-point square root double truncate
GF.SQR.64.X	Group floating-point square root double exact

	op	prec	round/trap
absolute value	ABS	16 32 64	NONE X
float from integer	FLOAT	16 32 64	NONE C F N T X
integer from float	SINK	16 32 64	NONE C F N T X
increase format precision	INFLATE	16 32	NONE X
decrease format precision	DEFLATE	32 64	NONE C F N T X
square root	SQR	16 32 64	NONE C F N T X

Format

GF.op.prec.round rc=ra

Description

The contents of register ra is used as the operand of the specified floating-point operation. The result is placed in register rc. The operation is rounded using the specified rounding option or using round-to-nearest if not specified. If a rounding option is specified, the operation raises a floating point exception if a floating-point invalid operation, divide by zero, overflow, or underflow occurs, or when specified, if the result is inexact. If a rounding option is not specified, floating-point exceptions are not raised, and are handled according to the default rules of IEEE 754.

Definition

```
def GroupFloatingPointUnary(op,prec,round,ra,rb,rc) as
    a ← REG[ra]
    case op of
        GF.ABS, GF.NEG, GF.SQR:
            for i ← 0 to 128-prec by prec
                ai ← F(prec,a(i+prec-1..i))
                case op of
                    GF.ABS:
                        if ai < 0 then
                            ci ← -ai
                        else
                            ci ← ai
                        endif
                    GF.NEG:
```

```
                    ci ← -ai
                GF.SQR:
                    ci ← √ai
            endcase
            ci+prec-1..i ← PackF(prec, ci, round)
        endfor
    GF.SINK:
        for i ← 0 to 128-prec by prec
            ai ← F(prec,ai+prec-1..i)
            ci+prec-1..i ← ai
        endfor
    GF.FLOAT:
        for i ← 0 to 128-prec by prec
            ai ← ai+prec-1..i
            ci+prec-1..i ← PackF(prec,ai, round)
        endfor
    GF.INFLATE:
        for i ← 0 to 64-prec by prec
            ai ← F(prec,ai+prec-1..i)
            ci+i+prec+prec-1..i+i ← PackF(prec+prec,ai, round)
        endfor
    GF.DEFLATE:
        for i ← 0 to 128-prec by prec
            ai ← F(prec,ai+prec-1..i)
            ci/2+prec/2-1..i/2 ← PackF(prec/2,ai, round)
        endfor
endcase
REC[rc] ← c
enddef
```

Exceptions

Reserved instruction
Floating-point arithmetic

Load

These operations add the contents of two registers to produce a virtual address, load data from memory, sign- or zero-extending the data to fill the destination register.

Operation codes

L.8[22]	Load signed byte
L.16.B	Load signed doublet big-endian
L.16.B.A	Load signed doublet big-endian aligned
L.16.L	Load signed doublet little-endian
L.16.L.A	Load signed doublet little-endian aligned
L.32.B	Load signed quadlet big-endian
L.32.B.A	Load signed quadlet big-endian aligned
L.32.L	Load signed quadlet little-endian
L.32.L.A	Load signed quadlet little-endian aligned
L.64.B[23]	Load octlet big-endian
L.64.B.A[24]	Load octlet big-endian aligned
L.64.L[25]	Load octlet little-endian
L.64.L.A[26]	Load octlet little-endian aligned
L.128.B[27]	Load hexlet big-endian
L.128.B.A[28]	Load hexlet big-endian aligned
L.128.L[29]	Load hexlet little-endian
L.128.L.A[30]	Load hexlet little-endian aligned
L.U.8[31]	Load unsigned byte
L.U.16.B	Load unsigned doublet big-endian
L.U.16.B.A	Load unsigned doublet big-endian aligned
L.U.16.L	Load unsigned doublet little-endian

[22] L.8 need not distinguish between little-endian and big-endian ordering, nor between aligned and unaligned, as only a single byte is loaded.
[23] L.64.B need not distinguish between signed and unsigned, as the octlet fills the destination register.
[24] L.64.B.A need not distinguish between signed and unsigned, as the octlet fills the destination register.
[25] L.64.L need not distinguish between signed and unsigned, as the octlet fills the destination register.
[26] L.64.L.A need not distinguish between signed and unsigned, as the octlet fills the destination register.
[27] L.128.B need not distinguish between signed and unsigned, as the hexlet fills the destination register pair.
[28] L.128.B.A need not distinguish between signed and unsigned, as the hexlet fills the destination register pair.
[29] L.128.L need not distinguish between signed and unsigned, as the hexlet fills the destination register pair.
[30] L.128.L.A need not distinguish between signed and unsigned, as the hexlet fills the destination register pair.
[31] L.U8 need not distinguish between little-endian and big-endian ordering, nor between aligned and unaligned, as only a single byte is loaded.

L.U.16.L.A	Load unsigned doublet little-endian aligned
L.U.32.B	Load unsigned quadlet big-endian
L.U.32.B.A	Load unsigned quadlet big-endian aligned
L.U.32.L	Load unsigned quadlet little-endian
L.U.32.L.A	Load unsigned quadlet little-endian aligned
L.U.64.B	Load unsigned octlet big-endian
L.U.64.B.A	Load unsigned octlet big-endian aligned
L.U.64.L	Load unsigned octlet little-endian
L.U.64.L.A	Load unsigned octlet little-endian aligned

number format	type	size	ordering	alignment
signed byte		8		
unsigned byte	U	8		
signed integer		16 32 64	L B	
signed integer aligned		16 32 64	L B	A
unsigned integer	U	16 32 64	L B	
unsigned integer aligned	U	16 32 64	L B	A
register		128	L B	
register aligned		128	L B	A

Format

op rc=ra,rb

Description

A virtual address is computed from the sum of the contents of register ra and register rb. The contents of memory using the specified byte order is treated as the size specified and zero-extended or sign-extended as specified, and placed into register rb.

If alignment is specified, the computed virtual address must be aligned, that is, it must be an exact multiple of the size expressed in bytes. If the address is not aligned an "access disallowed by virtual address" exception occurs.

Definition

MU 0023340

```
def Load(op,ra,rb,rc) as
    case op of
        L16L, L32L, L8, L16LA, L32LA, L16B, L32B, L16BA, L32BA,
        L64L, L64LA, L64B, L64BA:
            signed ← true
        LU16L, LU32L, LU8, LU16LA, LU32LA, LU16B, LU32B, LU16BA, LU32BA,
        LU64L, LU64LA, LU64B, LU64BA:
            signed ← false
        L128L, L128LA, L128B, L128BA:
```

```
                signed ← undefined
        endcase
        case op of
            L8, LU8:
                size ← 8
            L16L, LU16L, L16LA, LU16LA, L16B, LU16B, L16BA, LU16BA:
                size ← 16
            L32L, LU32L, L32LA, LU32LA, L32B, LU32B, L32BA, LU32BA:
                size ← 32
            L64L, LU64L, L64LA, LU64LA, L64B, LU64B, L64BA, LU64BA:
                size ← 64
            L128L, L128LA, L128B, L128BA:
                size ← 128
        endcase
        case op of
            L16L, LU16L, L32L, LU32L, L64L, LU64L, L128L,
            L16LA, LU16LA, L32LA, LU32LA, L64LA, LU64LA, L128LA:
                order ← L
            L16B, LU16B, L32B, LU32B, L64B, LU64B, L128B,
            L16BA, LU16BA, L32BA, LU32BA, L64BA, LU64BA, L128BA:
                order ← B
            L8, LU8:
                order ← undefined
        endcase
        case op of
            L16L, LU16L, L32L, LU32L, L64L, LU64L, L128L,
            L16B, LU16B, L32B, LU32B, L64B, LU64B, L128B:
                align ← false
            L16LA, LU16LA, L32LA, LU32LA, L64LA, LU64LA, L128LA,
            L16BA, LU16BA, L32BA, LU32BA, L64BA, LU64BA, L128BA:
                align ← true
            L8, LU8:
                align ← undefined
        endcase
        VirtAddr ← REG[ra] + REG[rb]
        if align then
            if (VirtAddr and ((size/8)-1)) ≠ 0 then
                raise AccessDisallowedByVirtualAddress
            endif
        endif
        m ← LoadMemory(VirtAddr,size,order)
        mx ← (msize-1 and signed)^(128-size) || m
        REG[rc] ← mx
    enddef
```

Exceptions

Reserved instruction
Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TLB
Access disallowed by local TLB
Access detail required by tag
Access detail required by local TLB
Access detail required by global TLB
Cache coherence intervention required by tag
Cache coherence intervention required by local TLB

MU 0023341

Cache coherence intervention required by global TLB
Local TLB miss
Global TLB miss

Load Immediate

These operations add the contents of a register to a sign-extended immediate value to produce a virtual address, load data from memory, sign- or zero-extending the data to fill the destination register.

MU 0023343

Operation codes

L.8.I[32]	Load signed byte immediate
L.16.B.A.I	Load signed doublet big-endian aligned immediate
L.16.B.I	Load signed doublet big-endian immediate
L.16.L.A.I	Load signed doublet little-endian aligned immediate
L.16.L.I	Load signed doublet little-endian immediate
L.32.B.A.I	Load signed quadlet big-endian aligned immediate
L.32.B.I	Load signed quadlet big-endian immediate
L.32.L.A.I	Load signed quadlet little-endian aligned immediate
L.32.L.I	Load signed quadlet little-endian immediate
L.64.B.A.I[33]	Load octlet big-endian aligned immediate
L.64.B.I[34]	Load octlet big-endian immediate
L.64.L.A.I[35]	Load octlet little-endian aligned immediate
L.64.L.I[36]	Load octlet little-endian immediate
L.128.B.A.I[37]	Load hexlet big-endian aligned immediate
L.128.B.I[38]	Load hexlet big-endian immediate
L.128.L.A.I[39]	Load hexlet little-endian aligned immediate
L.128.L.I[40]	Load hexlet little-endian immediate
L.U.8.I[41]	Load unsigned byte immediate
L.U.16.B.A.I	Load unsigned doublet big-endian aligned immediate
L.U.16.B.I	Load unsigned doublet big-endian immediate
L.U.16.L.A.I	Load unsigned doublet little-endian aligned immediate

[32] L.8.I need not distinguish between little-endian and big-endian ordering, nor between aligned and unaligned, as only a single byte is loaded.
[33] L.64.B.A.I need not distinguish between signed and unsigned, as the octlet fills the destination register.
[34] L.64.B.I need not distinguish between signed and unsigned, as the octlet fills the destination register.
[35] L.64.L.A.I need not distinguish between signed and unsigned, as the octlet fills the destination register.
[36] L.64.L.I need not distinguish between signed and unsigned, as the octlet fills the destination register.
[37] L.128.B.A.I need not distinguish between signed and unsigned, as the hexlet fills the destination register pair.
[38] L.128.B.I need not distinguish between signed and unsigned, as the hexlet fills the destination register pair.
[39] L.128.L.A.I need not distinguish between signed and unsigned, as the hexlet fills the destination register pair.
[40] L.128.L.I need not distinguish between signed and unsigned, as the hexlet fills the destination register pair.
[41] L.U8.I need not distinguish between little-endian and big-endian ordering, nor between aligned and unaligned, as only a single byte is loaded.

L.U.16.L.I	Load unsigned doublet little-endian immediate
L.U.32.B.A.I	Load unsigned quadlet big-endian aligned immediate
L.U.32.B.I	Load unsigned quadlet big-endian immediate
L.U.32.L.A.I	Load unsigned quadlet little-endian aligned immediate
L.U.32.L.I	Load unsigned quadlet little-endian immediate
L.U.64.B.A.I	Load unsigned octlet big-endian aligned immediate
L.U.64.B.I	Load unsigned octlet big-endian immediate
L.U.64.L.A.I	Load unsigned octlet little-endian aligned immediate
L.U.64.L.I	Load unsigned octlet little-endian immediate

number format	type	size	ordering	alignment
signed byte		8		
unsigned byte	U	8		
signed integer		16 32 64	L B	
signed integer aligned		16 32 64	L B	A
unsigned integer	U	16 32 64	L B	
unsigned integer aligned	U	16 32 64	L B	A
register		128	L B	
register aligned		128	L B	A

Format

op rb=ra,offset

Description

A virtual address is computed from the sum of the contents of register ra and the sign-extended value of the offset field. The contents of memory using the specified byte order is treated as the size specified and zero-extended or sign-extended as specified, and placed into register rb.

If alignment is specified, the computed virtual address must be aligned, that is, it must be an exact multiple of the size expressed in bytes. If the address is not aligned an "access disallowed by virtual address" exception occurs.

Definition

```
def LoadImmediate(op,ra,rb,offset) as
    case op of
        L16LI, L32LI, L8I, L16LAI, L32LAI, L16BI, L32BI, L16BAI, L32BAI:
        L64LI, L64LAI, L64BI, L64BAI:
            signed ← true
        LU16LI, LU32LI, LU8I, LU16LAI, LU32LAI,
        LU16BI, LU32BI, LU16BAI, LU32BAI:
        LU64LI, LU64LAI, LU64BI, LU64BAI:
            signed ← false
        L128LI, L128LAI, L128BI, L128BAI:
```

```
            signed ← undefined
    endcase
    case op of
        L8I, LU8I:
            size ← 8
        L16LI, LU16LI, L16LAI, LU16LAI, L16BI, LU16BI, L16BAI, LU16BAI:
            size ← 16
        L32LI, LU32LI, L32LAI, LU32LAI, L32BI, LU32BI, L32BAI, LU32BAI:
            size ← 32
        L64LI, LU64LI, L64LAI, LU64LAI, L64BI, LU64BI, L64BAI, LU64BAI:
            size ← 64
        L128LI, L128LAI, L128BI, L128BAI:
            size ← 128
    endcase
    case op of
        L16LI, LU16LI, L32LI, LU32LI, L64LI, LU64LI, L128LI,
        L16BI, LU16BI, L32BI, LU32BI, L64BI, LU64BI, L128BI:
            align ← false
        L16LAI, LU16LAI, L32LAI, LU32LAI, L64LAI, LU64LAI, L128LAI,
        L16BAI, LU16BAI, L32BAI, LU32BAI, L64BAI, LU64BAI, L128BAI:
            align ← true
        L8I, LU8I:
            align ← undefined
    endcase
    case op of
        L16LI, LU16LI, L32LI, LU32LI, L64LI, LU64LI, L128LI,
        L16LAI, LU16LAI, L32LAI, LU32LAI, L64LAI, LU64LAI, L128LAI:
            order ← L
        L16BI, LU16BI, L32BI, LU32BI, L64BI, LU64BI, L128BI,
        L16BAI, LU16BAI, L32BAI, LU32BAI, L64BAI, LU64BAI, L128BAI:
            order ← B
        L8I, LU8I:
            order ← undefined
    endcase
    VirtAddr ← REG[ra] + (offset11.52 || offset)
    if align then
        if (VirtAddr and ((size/8)-1)) ≠ 0 then
            raise AccessDisallowedByVirtualAddress
        endif
    endif
    b ← LoadMemory(VirtAddr,size,order)
    bx ← (bsize-1 and signed)128-size || b
    REG[rb] ← bx
enddef
```

Exceptions

Reserved instruction
Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TLB
Access disallowed by local TLB
Access detail required by tag
Access detail required by local TLB
Access detail required by global TLB
Cache coherence intervention required by tag
Cache coherence intervention required by local TLB

MU 0023345

Cache coherence intervention required by global TLB
Local TLB miss
Global TLB miss

Store

These operations add the contents of two registers to produce a virtual address, and store the contents of a register into memory.

Operation codes

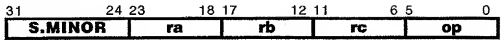
S.8[42]	Store byte
S.16.B	Store double big-endian
S.16.B.A	Store double big-endian aligned
S.16.L	Store double little-endian
S.16.L.A	Store double little-endian aligned
S.32.B	Store quadlet big-endian
S.32.B.A	Store quadlet big-endian aligned
S.32.L	Store quadlet little-endian
S.32.L.A	Store quadlet little-endian aligned
S.64.B	Store octlet big-endian
S.64.B.A	Store octlet big-endian aligned
S.64.L	Store octlet little-endian
S.64.L.A	Store octlet little-endian aligned
S.128.B	Store hexlet big-endian
S.128.B.A	Store hexlet big-endian aligned
S.128.L	Store hexlet little-endian
S.128.L.A	Store hexlet little-endian aligned
S.AAS.64.B.A	Store add-and-swap octlet big-endian aligned
S.AAS.64.L.A	Store add-and-swap octlet little-endian aligned
S.CAS.64.B.A	Store compare-and-swap octlet big-endian aligned
S.CAS.64.L.A	Store compare-and-swap octlet little-endian aligned
S.MAS.64.B.A	Store multiplex-and-swap octlet big-endian aligned
S.MAS.64.L.A	Store multiplex-and-swap octlet little-endian aligned
S.MUX.64.B.A	Store multiplex octlet big-endian aligned
S.MUX.64.L.A	Store multiplex octlet little-endian aligned

size				ordering		alignment
8						
16	32	64	128	L	B	
16	32	64	128	L	B	A

[42] S.8 need not specify byte ordering, nor need it specify alignment checking, as it stores a single byte.

Format

op ra,rb,rc



Description

A virtual address is computed from the sum of the contents of register ra and register rb. The contents of register rc, treated as the size specified, is stored in memory using the specified byte order.

If alignment is specified, the computed virtual address must be aligned, that is, it must be an exact multiple of the size expressed in bytes. If the address is not aligned an "access disallowed by virtual address" exception occurs.

Definition

```
def Store(op,ra,rb,rc) as
    case op of
        S8,
        S16L, S16LA, S16B, S16BA,
        S32L, S32LA, S32B, S32BA,
        S64L, S64LA, S64B, S64BA,
        S128L, S128LA, S128B, S128BA:
            function ← NONE
        SAAS64BA, SAAS64LA:
            function ← AAS
        SCAS64BA, SCAS64LA:
            function ← CAS
        SMAS64BA, SMAS64LA:
            function ← MAS
        SMUX64BA, SMUX64LA:
            function ← MUX
    endcase
    case op of
        S8:
            size ← 8
        S16L, S16LA, S16B, S16BA:
            size ← 16
        S32L, S32LA, S32B, S32BA:
            size ← 32
        S64L, S64LA, S64B, S64BA,
        SAAS64BA, SAAS64LA:
            size ← 64
        SCAS64BA, SCAS64LA, SMAS64BA, SMAS64LA, SMUX64BA, SMUX64LA:
            size ← 64
        S128L, S128LA, S128B, S128BA:
            size ← 128
    endcase
    case op of
        S8:
            align ← undefined
        S16L, S32L, S64L, S128L,
```

```
            S16B, S32B, S64B, S128B:
                align ← false
            S16LA, S32LA, S64LA, S128LA,
            S16BA, S32BA, S64BA, S128BA,
            SAAS64BA, SAAS64LA, SCAS64BA, SCAS64LA,
            SMAS64BA, SMAS64LA, SMUX64BA, SMUX64LA:
                align ← true
        endcase
        case op of
            S8:
                order ← undefined
            S16L, S32L, S64L, S128L,
            S16LA, S32LA, S64LA, S128LA,
            SAAS64LA, SCAS64LA, SMAS64LA, SMUX64LAI:
                order ← L
            S16B, S32B, S64B, S128B,
            S16BA, S32BA, S64BA, S128BA,
            SAAS64BA, SCAS64BA, SMAS64BA, SMUX64BAI:
                order ← B
        endcase
        VirtAddr ← REG[ra] + REG[rb]
        if align then
            if (VirtAddr and ((size/8)-1)) ≠ 0 then
                raise AccessDisallowedByVirtualAddress
            endif
        endif
        m ← REG[rc]
        case function of
            NONE:
                StoreMemory(VirtAddr,size,order,m_size-1..0)
            AAS:
                c ← LoadMemory(VirtAddr,size,order)
                StoreMemory(VirtAddr,size,order,m63..0+c)
                REG[rc] ← c
            CAS:
                c ← LoadMemory(VirtAddr,size,order)
                if (c = m63..0) then
                    StoreMemory(VirtAddr,size,order,m127..64)
                endif
                REG[rc] ← c
            MAS:
                c ← LoadMemory(VirtAddr,size,order)
                n ← (m127..64 & m63..0) | (c & ~m63..0)
                StoreMemory(VirtAddr,size,order,n)
                REG[rc] ← c
            MUX:
                c ← LoadMemory(VirtAddr,size,order)
                n ← (m127..64 & m63..0) | (c & ~m63..0)
                StoreMemory(VirtAddr,size,order,n)
        endcase
enddef
```

Exceptions

MU 0023349

Reserved instruction
Access disallowed by virtual address
Access disallowed by tag

Access disallowed by global TLB
Access disallowed by local TLB
Access detail required by tag
Access detail required by local TLB
Access detail required by global TLB
Cache coherence intervention required by tag
Cache coherence intervention required by local TLB
Cache coherence intervention required by global TLB
Local TLB miss
Global TLB miss

Store Immediate

These operations add the contents of a register to a sign-extended immediate value to produce a virtual address, and store the contents of a registerinto memory.

Operation codes

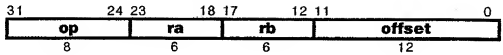
S.8.I[43]	Store byte immediate
S.16.B.A.I	Store double big-endian aligned immediate
S.16.B.I	Store double big-endian immediate
S.16.L.A.I	Store double little-endian aligned immediate
S.16.L.I	Store double little-endian immediate
S.32.B.A.I	Store quadlet big-endian aligned immediate
S.32.B.I	Store quadlet big-endian immediate
S.32.L.A.I	Store quadlet little-endian aligned immediate
S.32.L.I	Store quadlet little-endian immediate
S.64.B.A.I	Store octlet big-endian aligned immediate
S.64.B.I	Store octlet big-endian immediate
S.64.L.A.I	Store octlet little-endian aligned immediate
S.64.L.I	Store octlet little-endian immediate
S.128.B.A.I	Store hexlet big-endian aligned immediate
S.128.B.I	Store hexlet big-endian immediate
S.128.L.A.I	Store hexlet little-endian aligned immediate
S.128.L.I	Store hexlet little-endian immediate
S.AAS.64.B.A.I	Store add-and-swap octlet big-endian aligned immediate
S.AAS.64.L.A.I	Store add-and-swap octlet little-endian aligned immediate
S.CAS.64.B.A.I	Store compare-and-swap octlet big-endian aligned immediate
S.CAS.64.L.A.I	Store compare-and-swap octlet little-endian aligned immediate
S.MAS.64.B.A.I	Store multiplex-and-swap octlet big-endian aligned immediate
S.MAS.64.L.A.I	Store multiplex-and-swap octlet little-endian aligned immediate
S.MUX.64.B.A.I	Store multiplex octlet big-endian aligned immediate
S.MUX.64.L.A.I	Store multiplex octlet little-endian aligned immediate

size				ordering		alignment
8						
16	32	64	128	L	B	
16	32	64	128	L	B	A

[43] S.8.I need not specify byte ordering, nor need it specify alignment checking, as it stores a single byte.

Format

S.size.order.align.l ra,rb,offset



Description

A virtual address is computed from the sum of the contents of register ra and the sign-extended vale of the offset field. The contents of register rb, treated as the size specified, is stored in memory using the specified byte order.

If alignment is specified, the computed virtual address must be aligned, that is, it must be an exact multiple of the size expressed in bytes. If the address is not aligned an "access disallowed by virtual address" exception occurs.

Definition

```
def StoreImmediate(op,ra,rb,offset),as:
    case op of
        S8I,
        S16LI, S16LAI, S16BI, S16BAI,
        S32LI, S32LAI, S32BI, S32BAI,
        S64LI, S64LAI, S64BI, S64BAI,
        S128LI, S128LAI, S128BI, S128BAI:
            function ← NONE
        SAAS64BAI, SAAS64LAI:
            function ← AAS
        SCAS64BAI, SCAS64LAI:
            function ← CAS
        SMAS64BAI, SMAS64LAI:
            function ← MAS
        SMUX64BAI, SMUX64LAI:
            function ← MUX
    endcase
    case op of
        S8I:
            size ← 8
        S16LI, S16LAI, S16BI, S16BAI:
            size ← 16
        S32LI, S32LAI, S32BI, S32BAI:
            size ← 32
        S64LI, S64LAI, S64BI, S64BAI,
        SAAS64BAI, SAAS64LAI:
        SCAS64BAI, SCAS64LAI, SMAS64BAI, SMAS64LAI, SMUX64BAI, SMXU64LAI:
            size ← 64
        S128LI, S128LAI, S128BI, S128BAI:
            size ← 128
    endcase
    case op of
        S8I:
            align ← undefined
        S16LI, S32LI, S64LI, S128LI,
        S16BI, S32BI, S64BI, S128BI:
```

```
            align ← false
        S16LAI, S32LAI, S64LAI, S128LAI,
        S16BAI, S32BAI, S64BAI, S128BAI,
        SAAS64BAI, SAAS64LAI, SCAS64BAI, SCAS64LAI,
        SMAS64BAI, SMAS64LAI, SMUX64BAI, SMUX64LAI:
            align ← true
    endcase
    case op of
        S8I:
            order ← undefined
        S16LI, S32LI, S64LI, S128LI,
        S16LAI, S32LAI, S64LAI, S128LAI,
        SAAS64LAI, SCAS64LAI, SMAS64LAI, SMUX64LAI:
            order ← L
        S16BI, S32BI, S64BI, S128BI,
        S16BAI, S32BAI, S64BAI, S128BAI,
        SAAS64BAI, SCAS64BAI, SMAS64BAI, SMUX64BAI:
            order ← B
    endcase
    VirtAddr ← REG[ra] + (offset11^50 || offset)
    if align then
        if (VirtAddr and ((size/8)-1)) ≠ 0 then
            raise AccessDisallowedByVirtualAddress
        endif
    endif
    m ← REG[rb]
    case function of
        NONE:
            StoreMemory(VirtAddr,size,order,m_size-1..0)
        AAS:
            b ← LoadMemory(VirtAddr,size,order)
            StoreMemory(VirtAddr,size,order,m63..0+b)
            REG[rb] ← b
        CAS:
            b ← LoadMemory(VirtAddr,size,order)
            if (b = m63..0) then
                StoreMemory(VirtAddr,size,order,m127..64)
            endif
            REG[rb] ← b
        MAS:
            b ← LoadMemory(VirtAddr,size,order)
            n ← (m127..64 & m63..0) | (b & ~m63..0)
            StoreMemory(VirtAddr,size,order,n)
            REG[rb] ← b
        MUX:
            b ← LoadMemory(VirtAddr,size,order)
            n ← (m127..64 & m63..0) | (b & ~m63..0)
            StoreMemory(VirtAddr,size,order,n)
    endcase
enddef
```

MU 0023353

Exceptions

Reserved instruction
Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TLB

Access disallowed by local TLB
Access detail required by tag
Access detail required by local TLB
Access detail required by global TLB
Cache coherence intervention required by tag
Cache coherence intervention required by local TLB
Cache coherence intervention required by global TLB
Local TLB miss
Global TLB miss

Memory Management

This section discusses the caches, the translation mechanisms, the memory interfaces, and how the multiprocessor interface is used to maintain cache coherence.

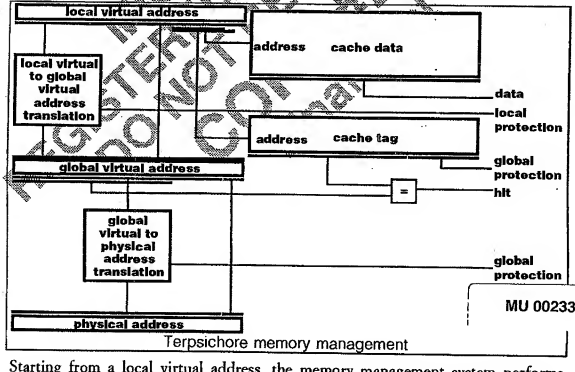
The Terpsichore processor provides for both local and global virtual addressing, arbitrary page sizes, and coherent-cache multiprocessors. The memory management system is designed to provide the requirements for implementation of virtual machines as well as virtual memory.

All facilities of the memory management system are themselves memory mapped, in order to provide for the manipulation of these facilities by high-level language, compiled code.

The translation mechanism is designed to allow full byte-at-a-time control of access to the virtual address space, with the assistance of fast exception handlers.

Privilege levels provide for the secure transition between insecure user code and secure system facilities. Instructions execute at a privilege specified by a two-bit field in the access information. Zero is the least-privileged level, and three is the most-privileged level.

The diagram below sketches the organization of the memory management system:



Terpsichore memory management

Starting from a local virtual address, the memory management system performs three actions in parallel: the low-order bits of the virtual address are used to directly access the data in the cache, a low-order bit field is used to access the

cache tag, and the high-order bits of the virtual address are translated from a local address space to a global virtual address space.

Following these three actions, operations vary depending upon the cache implementation. The cache tag may contain either a physical address and access control information (a physically-tagged cache), or may contain a global virtual address and global protection information (a virtually-tagged cache).

For a physically-tagged cache, the global virtual address is translated to a physical address by the TLB, which generates global protection information. The cache tag is checked against the physical address, to determine a cache hit. In parallel, the local and global protection information is checked.

For a virtually-tagged cache, the cache tag is checked against the global virtual address, to determine a cache hit, and the local and global protection information is checked. If the cache misses, the global virtual address is translated to a physical address by the TLB, which also generates the global protection information.

Local and Global Virtual Addresses

The 64-bit global virtual address space is global among all tasks. In a multitask environment, requirements for a task-local address space arise from operations such as the UNIX "fork" function, in which a task is duplicated into parent and child tasks, each now having a unique virtual address space. In addition, when switching tasks, access to one task's address space must be disabled and another task's access enabled.

Terpsichore provides for portions of the address space to be made local to individual tasks, with a translation to the global virtual space specified by four 16-bit registers for each local virtual space. Terpsichore specifies four sets of virtual spaces, and therefore four sets of these four registers. The registers specify a mask selecting which of the high-order 16 address bits are checked to match a particular value, and if they match, a value with which to modify the virtual address. Terpsichore avoids setting a fixed page size or local address size; these can be set by software conventions.

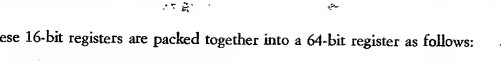
A local virtual address space is specified by the following::

field name	size	description
local mask	16	mask to select fields of local virtual address to perform match over
local match	16	value to perform match with masked local virtual address
local xor	16	value to xor with local virtual address if matched
local protect	16	local protection field (detailed later)

local virtual address space specifiers

These 16-bit registers are packed together into a 64-bit register as follows:

Local Translation Lookaside Buffer



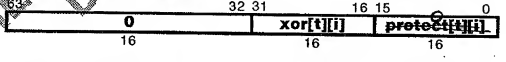
GTLB actual offset

The LTLB contains a separate context of register sets for each thread. A context consists of one or more sets of mask/match/xor/protect registers, one set for each simultaneously accessible local virtual address range. This set of registers is called the "Local TLB context," or LTLB (Local Translation Lookaside Buffer) context. The effect of this mechanism is to provide the facilities normally attributed to segmentation. However, in this system there is no extension of the address range, instead, segments are local nicknames for portions of the global virtual address space.

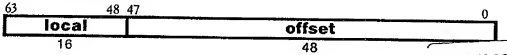
For instructions executing at the two least privileged levels (level 0 or level 1), a failure to match a LTLB entry causes an exception. This exception may be handled by loading an LTLB entry and continuing execution, thus providing access to an arbitrary number of local virtual address ranges.

Instructions executing at the two most privileged levels (level 2 or level 3) may access any region in the local virtual address space when a LTLB entry matches, and may acess regions in the global virtual address space when no LTLB entry matches. This mechanism permits privileged code to make judicious use of local virtual address ranges, which simplifies the manner in which privileged code may manipulate the contents of a local virtual address range on behalf of a less-privileged client.

A minimum implementation of an LTLB context is a single set of mask/match/xor/protect registers per thread. A single-set LTLB context may be further simplified by reserving the implementation of the mask and match registers, setting them to a read-only zero value:



If the largest possible space is reserved for an address space identifier, the virtual address is partitioned as shown below. Any of the bits marked as "local" below may be used as "offset" as desired.



MU 0023357

Definition

```
def GlobalVA,LocalProtect ← LocalVirtualToGlobalVirtualAddressTranslation(th,va,pl) as
    LocalTLBMatch ← NONE
```

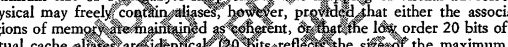
Highly Confidential

```
    for i ← 0 to <<sets per thread>>-1
        if (va63..48 & ~LocalTLB[th][i]63..48) = LocalTLB[th][i]47..32 then
            LocalTLBMatch ← i
        endif
    endfor
    if LocalTLBMatch = NONE then
        if pl < 2 then
            raise LocalTLBMiss
        endif
        GlobalVA ← va
        LocalProtect ← 0
    else
        GlobalVA ← (va63..48 ^ LocalTLB[th][LocalTLBMatch]31..16) || va47..0
        LocalProtect ← LocalTLB[th][LocalTLBMatch]15..0
    endif
enddef
```

Global Virtual Cache

The innermost levels of the instruction and data caches are direct-mapped and indexed and matched entirely by the global virtual address. Consequently, each block of memory data is tagged with access control information and the high-order bits of the global virtual address. The current size of the virtual caches is 32 kilobytes; for architectural compatibility, a minimum size of 8 kilobytes and a maximum size of 1 megabyte is specified. The mapping of virtual addresses to physical may freely contain aliases, however, provided that either the associated regions of memory are maintained as coherent, or that the low order 20 bits of any virtual cache aliases are identical. (20 bits reflects the size of the maximum 1M byte virtual cache.)

A virtual cache tag is contained in the buffer memory (described below). It is accessed in parallel with the virtual cache. The virtual tag must match, and the control information must permit the access, or a cache miss or exception occurs. There is one tag for each cache block; a cache block consists of 64 bytes, so for a 32 kilobyte cache, there is 4 kilobytes of cache tag information for each cache. The protect field shown below is the concatenation of the access, state and control fields shown in the table below:



Definition

The following function reads the data, tag, and protection bits from either the instruction (c=0) or data (c=1) cache, given a local virtual address.

```
def data,GlobalVA,GlobalProtect ← ReadCache(c,va,size) as
    data ← cacheDataArray[c][va14..4]
    GlobalVA ← cacheTagArray[c][va14..6]63..16 || va15..0
    GlobalProtect ← cacheTagArray[c][va14..6]15..0
enddef
```

MU 0023358

Translation and Protection

Global virtual addresses are translated to physical addresses only upon misses in the virtual caches. The translation is performed by software-programmable routines, augmented by a hardware TLB, specifically, the global TLB. The global TLB labels a cache line with the physical and access information in the virtual cache tag. The global TLB contains a minimum of 64 entries and a maximum of 256 entries.

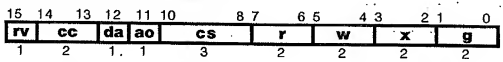
A local TLB, global TLB or virtual cache entry contains the following information. The figures in parentheses are the actual size of the field contained, if only a sub-field is held in the entry.

field name	size	description	local TLB	global TLB	cache tag
virtual address	64	virtual address (lowest address of region)	✔ (16)	✔ (58)	✔ (50)
virtual address mask	64	mask for virtual address match	✔ (16)	✔ (58)	
physical address	64	physical address xor virtual address	✔ (16)	✔ (58)	
reserved		additional space in register	✔ (2)	✔ (2)	✔ (1)
caching control	2	are accesses to this region incoherent (0), coherent (1), no-allocate (2), or uncached-physical(3)?	✔	✔	✔ (1)
detail access	1	do portions of this region have access controlled more restrictively?	✔	✔	✔
access ordering	1	are accesses to this region ordered weakly (0) or strongly (1) with respect to other accesses?	✔	✔	✔
coherence state	3	if region is coherently cached, does coherence state permit read(4), write(2), or replacement(1)? if region is not coherently cached, does cache state require write-back(2) or not(0)?	✔	✔	✔
read access	2	minimum privilege for read access	✔	✔	✔
write access	2	minimum privilege for write access	✔	✔	✔
execute access	2	minimum privilege for execute access	✔	✔	✔
gateway access	2	minimum privilege for gateway access	✔	✔	✔

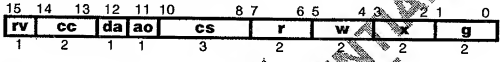
information in local TLB, global TLB, or virtual cache entry

The bottom section of the table above indicates the contents of the 16-bit protection field:

Protection information in local TLB



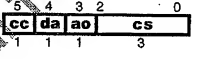
Protection information in global TLB



Protection information in instruction cache tag

Protection information in virtual data cache tag

Protection information in physical data cache tag



Memory Interface

Dedicated hardware mechanisms are provided to fetch and store back data blocks in the instruction and data caches, provided that a matching entry can be found in the global TLB. When no entry is to be found in the global TLB, an exception handler is invoked either to generate the required information from the virtual address, or to place an entry in the global TLB to provide for automatic handling of this and other similarly addressed data blocks.

The initial implementation of Terpsichore partitions the remainder of the local memory system, including a second-level joint physical cache and a DRAM-based memory array, into a set of separate devices, called Mnemosyne. These devices are accessed via a high-bandwidth, byte-wide packet interface, called Hermes, which is largely transparent to the Terpsichore architecture. The Mnemosyne devices provide single-bit correction, double-bit detection ECC on all local memory accesses, and a check byte protects all packet interface transfers. The

size of the secondary cache and the DRAM memory array is implementation-dependent.

Cache Coherence

The Terpsichore processor is intended for use in either a uniprocessor or multiprocessor environment. At the high performance level intended for Terpsichore, mechanisms employed in other processor designs to maintain cache coherence, such as "snoopy cache" buses, cannot be effectively built, because the communication latency and bandwidth between processors would be excessive. Several cache coherence mechanisms have been designed for high-performance processors that do not require that all memory transactions be broadcast among the processors in a system, one of which is the Scalable Coherent Interface, or SCI, as specified by the IEEE Standard 1596-1992.

The SCI cache coherence mechanism is extremely complex. Many of the cache coherence operations take time proportional to the number of processors involved in the operation, and so implicitly assume that the number of processors sharing a particular data item will tend to be much less than the 64k processors that SCI can theoretically handle (SCI is now considering an even more complex mechanism that may assure logarithmic growth in time complexity). Most importantly, no complete working prototype of this mechanism has been built, tested and benchmarked at the time of development of the Terpsichore architecture.

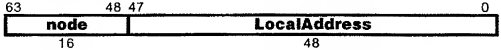
It is considered then, that the cache coherence mechanism should not be implemented in immutable, hardware state machines. A software implementation of a cache coherence protocol is proposed, which given the high intrinsic performance of the processor, is likely to reach nearly the performance level that can be achieved in dedicated hardware. The greatest advantage, though, is that Terpsichore will be an excellent vehicle to test and improve the performance of experimental non-bus-oriented cache coherency operations.

Cache coherence information is available within the local TLB, global TLB and the cache tags, so that coherence operations may be performed at task-level, page-level, or cache-block-level as desired. This flexibility provides for a coherent view of memory in multiprocessing systems with varying degrees of coupling.

Physical Addresses

MU 0023362

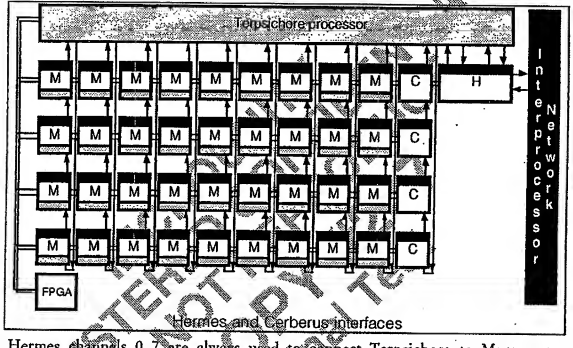
Physical addresses are 64 bits in size, consisting of a 16 bit processor node number and a 48 bit address.



Physical addresses in which the node number is zero reference the local processors local memory space, providing access to local memory, cache tags, system and interface facilities. Physical addresses in which the node number is

nonzero reference other processors' local memory spaces, using the Hydra interface for communication.

The local memory environment of Terpsichore involves the use of up to twelve Hermes byte-wide packet communications channels, by which Terpsichore can request read or write transactions to Mnemosyne, Calliope, and Hydra devices. In addition, Terpsichore can issue read or write transactions to the Cerberus serial bus interface, via which the Mnemosyne, Calliope, Hydra and other devices' configuration and control registers can be accessed. The diagram illustrates shows one possible Terpsichore memory environment:



Hermes channels 0..7 are always used to connect Terpsichore to Mnemosyne memory devices. Hermes channels 8..11 may be used to connect Mnemosyne, Calliope, or Hydra devices.

Terpsichore provides three different mappings of the local memory environment in the local physical address space. The non-interleaved space provides for the access of all Mnemosyne, Calliope, and Hydra device memory spaces such that each device appears as a single continuous space. The uniprocessor spaces provide for the interleaved access of one, two, or four sets of eight Mnemosyne devices on separate Hermes channels as a single continuous space. The multiprocessor spaces provide for the interleaved access of one, two, or four sets of nine Mnemosyne devices on separate Hermes channels as a single continuous space with the ninth channel used as a cache coherency directory.

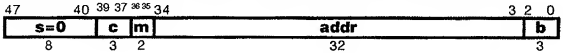
The value of the space field determines the interpretation of the 48-bit local address field as given by the following table:

value	interpretation
0	non-interleaved Hermes channel 0..7 space
1	non-interleaved Hermes channel 8..11 space
2	8x1-way interleaved uniprocessor memory space
3	9x1-way interleaved multiprocessor memory space
4	8x2-way interleaved uniprocessor memory space
5	9x2-way interleaved multiprocessor memory space
6	8x4-way interleaved uniprocessor memory space
7	9x4-way interleaved multiprocessor memory space
8	
9	
10	4x1-way interleaved uniprocessor memory space
11	5x1-way interleaved multiprocessor memory space
12	4x2-way interleaved uniprocessor memory space
13	5x2-way interleaved multiprocessor memory space
14	4x4-way interleaved uniprocessor memory space
15	9x4-way interleaved multiprocessor memory space
16	
17	
18	2x1-way interleaved uniprocessor memory space
19	3x1-way interleaved multiprocessor memory space
20	2x2-way interleaved uniprocessor memory space
21	3x2-way interleaved multiprocessor memory space
22	2x4-way interleaved uniprocessor memory space
23	3x4-way interleaved multiprocessor memory space
24..31	Cerberus memory and control space
32..255	

Space field interpretation

Non-interleaved Hermes channel 0..7 Space

The non-interleaved Hermes channel 0..7 space provides a single continuous memory space for each device in Hermes channels 0..7. Mnemosyne protocols are used. Only incoherent accesses are supported (no memory directory tags).



MU 0023364

The range of valid values and the interpretation of the fields is given by the following table:

field	value	interpretation
s	0	Specify non-interleaved Hermes channel 0..7 space
c	0..7	Hermes channels 0..7
m	0..3	Module address
addr	0..$2^{32}-1$	Logical memory block address
b	0..7	Pad for conversion of byte address to block address

non-interleaved space field interpretation

Non-interleaved Hermes channel 8..11 space

The non-interleaved Hermes channel 8..11 space provides a single continuous memory space for each device in Hermes channels 8..11. Either Mnemosyne or Calliope/Hydra protocols may be specified. Only incoherent accesses are supported (no memory directory tags).

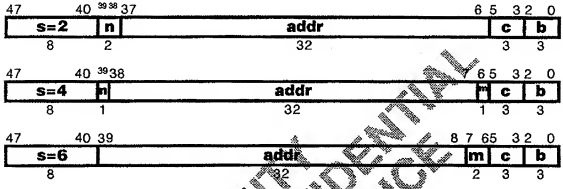
The range of valid values and the interpretation of the fields is given by the following table:

field	value	interpretation
s	1	Specify non-interleaved Hermes channel 8..11 space
h	0..1	0: use Mnemosyne protocol 1: use Calliope/Hydra protocol
c	0..3	Hermes channels 8..11
m	0..3	Module address
addr	0..$2^{32}-1$	Logical memory block address
b	0..7	Pad for conversion of byte address to block address

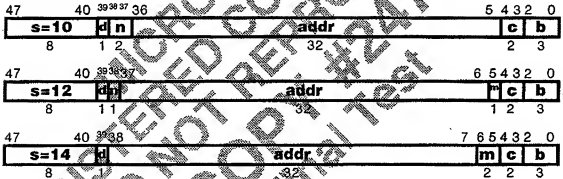
non-interleaved space field interpretation

Uniprocessor Interleaved Spaces

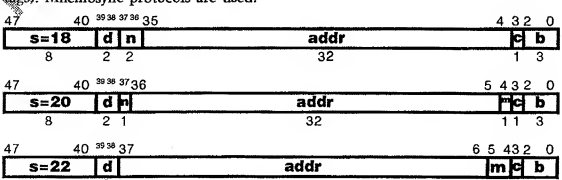
The interleaved spaces described below interleave between 8 Hermes channels (0..7), supporting only incoherent accesses (no memory directory tags). Mnemosyne protocols are used.



The interleaved spaces described below interleave between 4 Hermes channels (0..3 or 4..7), supporting only incoherent accesses (no memory directory tags). Mnemosyne protocols are used.



The interleaved spaces described below interleave between 2 Hermes channels (0..1, 2..3, 4..5, or 6..7), supporting only incoherent accesses (no memory directory tags). Mnemosyne protocols are used.



The range of valid values and the interpretation of the fields is given by the following table:

field	value	interpretation
s	2,4,6,10, 12,14,18, 20,22	Specify uniprocessor interleaved space
d	0..3	High-order bits of Hermes channel number
c	0..7	Low-order bits of Hermes channels number
n	0..3	High-order bits of Hermes module address
m	0..3	Low-order bits of Hermes module address
addr	$0..2^{32}-1$	Logical memory block address
b	0..7	Pad for conversion of byte to block address

interleaved space field interpretation

Hermes byte address

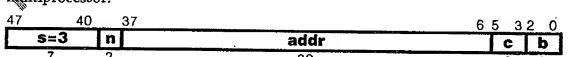
The Hermes channel number is constructed by concatenating the **d** and **c** fields:

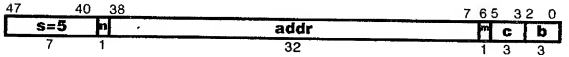
The Hermes module number is constructed by concatenating the **n** and **m** fields:

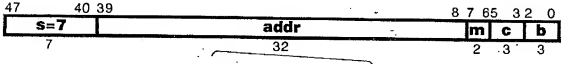
MU 0023367

Multiprocessor Interleaved Space

The interleaved spaces described below interleave between 9 channels for a multiprocessor.







Highly Confidential

The range of valid values and the interpretation of the fields is given by the following table:

field	value	interpretation
s	3, 5,7	Specify interleaved space
c	0..7	Mnemosyne channels 0..7, before modification described below
n	0..3	High-order bits of Hermes module address
m	0..3	Low-order bits of Hermes module address
addr	$0..2^{32}-1$	Logical memory block address
b	0..7	Pad for conversion of byte to block address

Interleaved space field interpretation

For the multiprocessor space the channel number field is modified by the low-order memory block address bits according to the following tables. In addition, access to a location in this space also accesses a memory tag, using the Hermes channel specified in the tables below.

addr$_{2..0}$	**c**								tag
	0	1	2	3	4	5	6	7	
0	8	1	2	3	4	5	6	7	0
1	4	5	6	7	0	8	2	3	1
2	8	3	0	1	6	7	4	5	2
3	6	7	4	5	2	8	0	1	3
4	2	0	3	2	5	8	7	6	4
5	8	4	7	6	1	0	3	2	5
6	3	2	1	0	7	8	5	4	6
7	8	6	5	4	3	2	1	0	7

9 channel translation table

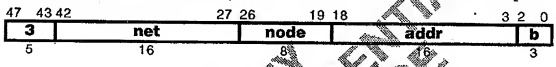
The memory tag entry is an octlet value for each 64-byte memory block. The contents of the tag is interpreted by Terpsichore hardware to signify (0) a zero value indicates that the memory block is not contained in the cache, (1) a value equal to the virtual address used to access the memory block indicates that the value is cached at that address, and (2) any other values indicates that the value may be cached in multiple or remote locations and requires software intervention for interpretation.

Thus a read to a memory block accesses the tag, and if the value is zero, fills it with the virtual address via which the access occurred. When the memory block is returned to memory, the tag is accessed, and if the value is equal to the virtual address, the tag is reset to zero. In all other cases, an exception occurs, which is handled by software to implement the cache coherency mechanisms.

We also need to have a space available in which access to the tag via software routines is straightforward - the non-interleaved space makes the tag available, but not conveniently.

SerialBus Space

The Cerberus serial bus space provides access to a memory space in which Bootstrap ROM code, Terpsichore, Mnemosyne and Calliope configuration data, and other Cerberus peripherals are accessed. The Cerberus serial bus is specified by the document: "Cerberus Serial Bus Architecture." Terpsichore configuration data is accessable via Cerberus as a slave device as well as via this address space.



The range of valid values and the interpretation of the fields is given by the following table:

field	value	interpretation
3	3	Specify Cerberus space
net	0..216-1	Specify Cerberus net address
node	0..255	Cerberus node address
addr	0..216-1	Logical memory block address
b	0..7	Pad for conversion of byte address to block address

Cerberus space field interpretation

Control Register Addresses

This section is under construction.

Local TLB (4 octlets)
Virtual cache tags (2k-128k x 2 caches)
Virtual cache data (16k-1M x 2 caches)
Global TLB (4 octlets x 64-256 entries=2k-8k bytes)

Events and Threads

Exceptions signal several kinds of events: (1) events that are indicative of failure of the software or hardware, such as arithmetic overflow or parity error, (2) events that are hidden from the virtual process model, such as translation buffer misses, (3) events that infrequently occur, but may require corrective action, such as floating-point underflow. In addition, there are (4) external events that cause scheduling of a computational process, such as completion of a disk transfer or clock events.

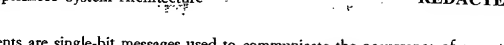
Each of these types of events require the interruption of the current flow of execution, handling of the exception or event, and in some cases, descheduling of the current task and rescheduling of another. The Terpsichore processor provides a mechanism that is based on the multi-threaded execution model of Mach. Mach divides the well-known UNIX process model into two parts, one called a task, which encompasses the virtual memory space, file and resource state, and the other called a thread, which includes the program counter, stack space, and other register file state. The sum of a Mach task and a Mach thread exactly equals one UNIX process, and the Mach model allows a task to be associated with several threads. On one processor at any one moment in time, at least one task with one thread is running.

In the taxonomy of events described above, the cause of the event may either be synchronous to the currently running thread, generally types 1, 2, and 3, or asynchronous and associated with another task and thread that is not currently running, generally type 4. For synchronous events, Terpsichore will suspend the currently running thread in the current task and continue execution with another thread that is dedicated to the handling of events. For asynchronous events, Terpsichore will continue execution with the dedicated event thread, while not necessarily suspending the currently running thread.

Terpsichore provides sufficient resources for the interleaved execution of at least one full thread, containing 64 general registers and a program counter, and at least one separate event thread, containing 16 general registers and a program counter. When both threads are able to continue execution, priority is generally given to the event threads.

All facilities of the exception, memory management, and interface systems are themselves memory mapped, in order to provide for the manipulation of these facilities by high-level language, compiled code. In particular, the thread resources of the full threads are memory-mapped so that the exception threads are able to read and write the general registers and program counter of the full threads.

Events are single-bit messages used to communicate the occurrence of exceptions between full threads and event threads and interface devices.



The event register appears at several locations in memory, with slightly different side effects on read and write operations.

offset	side effect on read	side effect on write
0	none: return event register contents	normal: write data into event register
8	stall thread until contents of event register is non-zero, then return event register contents	stall thread until bitwise and of data and event register contents is non-zero
16	return zero value (so read-modify-write for byte/doublet/quadlet store works)	one bits in data set (to one) corresponding event register bits
24	return zero value (so read-modify-write for byte/doublet/quadlet store works)	one bits in data clear (to zero) corresponding event register bits

interface devices signal events by responding to non-blocking read requests generated by a write to a Terpsichore control register. The response to these read requests is combined into the event register with an inclusive-or operation.

A write to the event daemon address register causes Terpsichore to issue a read request to the corresponding physical address. The device referenced by this request may respond at any future time with a value, which is inclusive-or'ed into the event register.

The following notes list the resources needed to support the threads...

Events:

0	full thread 0 suspended at instruction fetch because of exception
1	full thread 0 suspended at data fetch because of exception
2	full thread 0 suspended at execution because of exception
3	full thread 0 suspended at execution because of empty pipeline
4-15	same for full threads 1-3.
16-63	timer, calliope, and hydra events

Thread resources:
General registers at data fetch stage
General registers at execution state
Program counter, privilege level at data fetch stage
Program counter, privilege level at execution stage
Mask register: events which permit the thread to run
Mask register: events which prevent the thread to run?
Control register: suspend ifetch, dfetch, execute, thread priority
Local TLB entries (full threads only)
Exception information registers:
    exception cause
    instruction which caused exception
    virtual address at which access attempted
    size of access attempted
    type of access (read, write, execute, gateway)
    did exception occur at lower or higher addr if cross boundary?
don't need - register contents (can get from mem map GR)

Sort by stage:
Inst fetch stage:
    program counter
    exception state: cause: TLB miss, coherence action required...
    control register:
        suspend (drain queue),
        reset(clear queue)
        proceed past detail
Data fetch stage:
    General registers
    program counter, privilege level
    control register:
        suspend (drain queue),
        reset (clear queue)
        proceed past detail
    exception state: cause(incl access type, size, boundary,
        local TLB hit indication), inst
    can compute local va from GR, inst:
        shift-and-add-load-shiftl-shiftr-add (7)
    computing global va is hard...=> need global va register
    can compute size from inst, 16-word table: shift and add load (4)
    prefetched data, instruction queue
        clear queue
        drain queue
Execute stage:
    General registers
    program counter, privilege level
    control register: suspend
    exception state: cause(flt/fix arithmetic), inst

Exceptions:

MU 0023372

Highly Confidential

number	exception
0	Access disallowed by tag
1	Access detail required by tag
2	Cache coherence action required by tag
3	Access disallowed by virtual address
4	Access disallowed by global TLB
5	Access detail required by global TLB
6	Cache coherence action required by global TLB
7	Global TLB miss
8	Access disallowed by local TLB
9	Access detail required by local TLB
10	Cache coherence action required by local TLB
11	Local TLB miss
12	Floating-point arithmetic
13	Fixed-point arithmetic
14	Reserved instruction
15	

Parameter passing

There are no special registers to indicate details about the exception, such as the virtual address at which an access was attempted, or the operands of a floating-point operation that results in an exception. Instead, this information is available via memory-mapped registers.

When a synchronous exception occurs in a full thread, the corresponding thread's state is frozen, and a general event is signalled. An event thread should handle the exeception, in whatever manner is required, and then may restart the full thread by writing to the full thread's control register.

When a synchronous exception occurs in an event thread, an immediate transfer of control occurs to the machine check vector address, with information about the exception available in the **machine check cause** field of the **status register**. The transfer of control may overwrite state that may be necessary to recover from the exception; the intent is to provide a satisfactory post-mortem indication of the characteristics of the failure.

Exceptions in detail

This section is under construction. Terpsichore has changed from passing the parameters in registers to passing the parameters in memory-mapped registers, and the information in this section doesn't reflect the changes yet.

This section describes in detail the conditions under which exception occurs, the parameters passed to the exeception handler, and the handling of the result of the procedure.

Access disallowed by tag

This exception occurs when a read (load), write (store), execute, or gateway attempts to access a virtual address for which the matching virtual cache entry does not permit this access.

Prototype

int AccessDisallowedByTag(int address, int size, int access)

Description

The address at which the access was attempted is passed as *address*. The size of the access in bytes is passed as *size*. The type of access is passed as *access*, with 0 meaning read, 1 meaning write, 2 meaning execute, and 3 meaning gateway. The exception handler should determine accessibility, modify the virtual memory state if desired, and return if the access should be allowed. Upon return, execution is restarted and the access will be retried.

Access detail required by tag

This exception occurs when a read (load), write (store), or execute attempts to access a virtual address for which the matching virtual cache entry would permit this access, but the detail bit is set.

Prototype

int AccessDetailRequiredByTag(int address, int size, int access)

Description

The address at which the access was attempted is passed as *address*. The size of the access in bytes is passed as *size*. The type of access is passed as *access*, with 0 meaning read, 1 meaning write, 2 meaning execute, and 3 meaning gateway. The exception handler should determine accessibility and return if the access should be allowed. Upon return, execution is restarted and the access will be retried. If the detail bit is set in the matching virtual cache entry, access will be permitted.

Cache coherence action required by tag

This exception occurs when a read (load, execute, or gateway), write (store), or replacement attempts to access a virtual address for which the coherence state of the matching virtual cache entry cannot permit this access.

Prototype

int CacheCoherence InterventionRequired(int address, int size, int access)

Description

The address at which the access was attempted is passed as *address*. The size of the access in bytes is passed as *size*. The type of access is passed as *access*, with 0 meaning read, 1 meaning write, 2 meaning replacement. The exception handler should modify the cache status to make the cache line accessible. Upon return, execution is restarted and the access will be retried.

Access disallowed by global TLB

This exception occurs when a read (load), write (store), execute, or gateway attempts to access a virtual address for which the matching global TLB entry does not permit this access.

Prototype

int AccessDisallowedByGlobalTLB(int address, int size, int access)

Description

The address at which the access was attempted is passed as *address*. The size of the access in bytes is passed as *size*. The type of access is passed as *access*, with 0 meaning read, 1 meaning write, 2 meaning execute, and 3 meaning gateway. The exception handler should determine accessibility, modify the virtual memory state if desired, and return if the access should be allowed. Upon return, execution is restarted and the access will be retried. If the detail bit is set in the matching global TLB entry, access will be permitted.

Access detail required by global TLB

This exception occurs when a read (load), write (store), execute, or gateway attempts to access a virtual address for which the matching global TLB entry would permit this access, but the detail bit in the global TLB entry is set.

Prototype

int AccessDetailRequiredByGlobalTLB(int address, int size, int access)

Description

The address at which the access was attempted is passed as *address*. The size of the access in bytes is passed as *size*. The type of access is passed as *access*, with 0 meaning read, 1 meaning write, 2 meaning execute, and 3 meaning gateway. The exception handler should determine accessibility and return if the access should be allowed. Upon return, execution is restarted and the access will be forced to be permitted. If the access is not to be allowed, the handler should not return.

Cache coherence action required by global TLB

This exception occurs when a read (load, execute, or gateway), write (store), or replacement attempts to access a virtual address for which the coherence state of the matching global TLB entry cannot permit this access.

Prototype

int CacheCoherence InterventionRequired(int address, int size, int access)

Description

The address at which the access was attempted is passed as *address*. The size of the access in bytes is passed as *size*. The type of access is passed as *access*, with 0 meaning read, 1 meaning write, 2 meaning replacement. The exception handler should modify the virtual memory state to make the global TLB accessible. Upon return, execution is restarted and the access will be retried.

Global TLB miss

This exception occurs when a read (load), write (store), execute, or gateway attempts to access a virtual address for which no global TLB entry matches.

Prototype

void GlobalTLBMiss(int address, int size, int access)

Description

The address at which the global TLB miss occurred is passed as *address*. The size of the access in bytes is passed as *size*. The type of access is passed as *access*, with 0 meaning read, 1 meaning write, 2 meaning execute, and 3 meaning gateway. The exception handler should load a global TLB entry which defines the translation and protection for this address. Upon return, execution is restarted and the global TLB access will be attempted again.

Access disallowed by local TLB

This exception occurs when a read (load), write (store), execute, or gateway attempts to access a virtual address for which the matching local TLB entry does not permit this access.

Prototype

int AccessDisallowedByLocalTLB(int address, int size, int access) MU 0023376

Description

The address at which the access was attempted is passed as *address*. The size of the access in bytes is passed as *size*. The type of access is passed as *access*, with 0

meaning read, 1 meaning write, 2 meaning execute, and 3 meaning gateway. The exception handler should determine accessibility, modify the virtual memory state if desired, and return if the access should be allowed. Upon return, execution is restarted and the access will be retried.

Access detail required by local TLB

This exception occurs when a read (load), write (store), execute, or gateway attempts to access a virtual address for which the matching local TLB entry would permit this access, but the detail bit in the local TLB entry is set.

Prototype

int AccessDetailRequiredByLocalTLB(int address, int size, int access)

Description

The address at which the access was attempted is passed as *address*. The size of the access in bytes is passed as *size*. The type of access is passed as *access*, with 0 meaning read, 1 meaning write, 2 meaning execute, and 3 meaning gateway. The exception handler should determine accessibility and return if the access should be allowed. Upon return, execution is restarted and the access will be forced to be permitted. If the access is not to be allowed, the handler should not return.

Cache coherence action required by local TLB

This exception occurs when a read (load, execute, or gateway), write (store), or replacement attempts to access a virtual address for which the coherence state of the matching local TLB entry cannot permit this access.

Prototype

int CacheCoherenceInterventionRequired(int address, int size, int access)

Description

The address at which the access was attempted is passed as *address*. The size of the access in bytes is passed as *size*. The type of access is passed as *access*, with 0 meaning read, 1 meaning write, 2 meaning replacement. The exception handler should modify the virtual memory state to make the local TLB accessible. Upon return, execution is restarted and the access will be retried.

MU 0023377

Local TLB miss

This exception occurs when a read (load), write (store), execute, or gateway attempts to access a virtual address for which no local TLB entry matches.

Prototype

void LocalTLBMiss(int address, int size, int access)

Highly Confidential

Description

The address at which the local TLB miss occurred is passed as *address*. The size of the access in bytes is passed as *size*. The type of access is passed as *access*, with 0 meaning read, 1 meaning write, 2 meaning execute, and 3 meaning gateway. The exception handler should load a local TLB entry which defines the translation and protection for this address. Upon return, execution is restarted and the local TLB access will be attempted again.

Floating-point arithmetic

Prototype

quad FloatingPointArithmetic(int inst, quad ra, quad rb, quad rc)

Description

The contents of the instruction which was the cause of the exception is passed as *inst*, and the contents of registers ra, rb and rc are passed as *ra, rb* and *rc*. The exception handler should attempt to perform the function specified in the instruction and service any exceptional conditions that occur. The result of the function is placed into register rc or rd upon return.

Fixed-point arithmetic

Prototype

int FixedPointArithmetic(int inst, int ra, int rb)

Description

The contents of the instruction which was the cause of the exception is passed as *inst*, and the contents of registers ra and rb are passed as *ra* and *rb*. The exception handler should attempt to perform the function specified in the instruction and service any exceptional conditions that occur. The result of the function is placed into register rb or rc.

Reserved Instruction

Prototype:

MU 0023378

int ReservedInstruction(int inst, int ra, int rb)

Description:

The contents of the instruction which was the cause of the exception is passed as *inst*, and the contents of registers ra and rb are passed as *ra* and *rb*. The result of the function is placed into register rd.

Access Disallowed by virtual address

This exception occurs when a load, store, branch, or gateway refers to an aligned memory operand with an improperly aligned address.

Prototype:

int AccessDisallowedByVirtualAddress(int inst, int address)

Description:

The contents of the instruction which was the cause of the exception is passed as *inst*, and the address at which the access was attempted is passed as *address*.

Clock

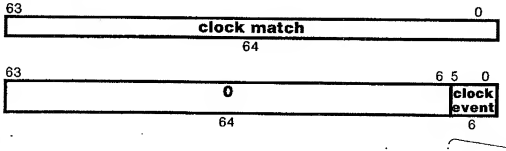
Each Euterpe processor includes a clock that maintains processor-clock-cycle accuracy. The value of the **clock cycle** register is incremented on every cycle, regardless of the number of instructions executed on that cycle. The **clock cycle** register is 64-bits long.

For testing purposes the clock cycle register is both readable and writable, though in normal operation it should be written only at system initialization time; there is no mechanism provided for adjusting the value in the clock cycle counter without the possibility of losing cycles.

Clock Event

An event is asserted when the value in the **clock cycle** register is equal to the value in the **clock match** register, which sets the specified **clock event** bit in the event register.

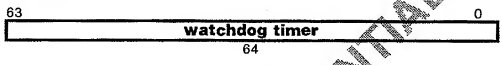
For testing purposes the **clock match** register is both readable and writable, though in normal operation it is normally written to.



Watchdog Timer

A Machine Check is asserted when the value in the **clock cycle** register is equal to the value in the **watchdog timer** register.

The **watchdog timer** register is both readable and writable, though in normal operation it is usually and periodically written with a sufficiently large value that the register does not equal the value in the **clock cycle** register before the next time it is written.

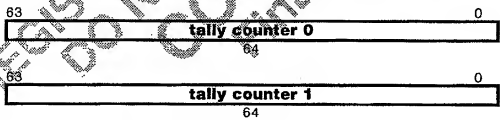


Tally Counter

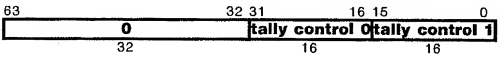
Each Enterpe processor includes two counters that can tally processor-related events or operations. The value of the **tally counter** registers are incremented on each processor clock cycle in which specified events or operations occur. The **tally counter** registers do not signal Enterpe events.

It is required that a sufficient number of bits be implemented so that the tally counter registers overflow no more frequently than once per second. 32 bits is sufficient for a 4 GHz clock. The remaining unimplemented bits must be zero whenever read, and ignored on write.

For testing purposes each of the tally counter registers are both readable and writable, though in normal operation each should be written only at system initialization time; there is no mechanism provided for adjusting the value in the event counter registers without the possibility of losing counts.



The tally counter control register selects one event for each of the event counters to tally.



The valid values for the tally control fields are given by the following table:

value	interpretation
0..63	tally events 0..63
64	freeze counter: count nothing
65	tally instructions processed by address unit
66	tally instructions processed by execute unit
67	tally instruction cache misses
68	tally data cache misses
69	tally data cache references
70..65535	Reserved

tally control field interpretation

Control Register Addresses

This section is under construction. Software and hardware designers should not infer anything about the value of the addresses from the ordering or entries in the tentative table below:

physical address	description
	event register
	event register with stall
	event register bit set
	event register bit clear
	event daemon address
	full thread 0, general register 0, data fetch stage
	...
	full thread 0, general register 63, data fetch stage
	full thread 0, program counter and privilege level, data fetch state
	full thread 0, control register, data fetch state
	full thread 0, status register, data fetch state
	full thread 0, general register 0, execution stage
	...
	full thread 0, general register 63, execution stage
	full thread 0, program counter and privilege level, execution state
	full thread 0, control register, execution stage
	full thread 0, status register, execution stage
	event thread 0, general register 0, data fetch stage
	...
	event thread 0, general register 15, data fetch stage
	event thread 0, program counter and privilege level, data fetch state
	event thread 0, control register, data fetch state
	event thread 0, status register, data fetch state
	event thread 0, general register 0, execution stage
	...
	event thread 0, general register 15, execution stage
	event thread 0, program counter and privilege level, execution state
	event thread 0, control register, execution stage
	event thread 0, status register, execution stage
	local TLB entry 0
	local TLB entry 1
	local TLB entry 2
	local TLB entry 3
	clock cycle register
	clock match register
	clock event control register
	tally counter 0
	tally counter 1
	tally control 0/1

Reset and Error Recovery

Certain external and internal events cause the Euterpe processor to invoke reset or error recovery operations. These operations consist of a full or partial reset of critical machine state, including initialization of the event thread to begin fetching instructions from the start vector address. Software may determine the nature of the reset or error by reading the value of the Cerberus control register, in which finding the **reset** bit set (1) indicates that a reset has occurred, finding the **clear** bit set (1) and the **reset** bit cleared (0) indicates that a logic clear has occurred, and finding both the **reset** and **clear** bits cleared (0) indicates that a machine check has occurred. When either a reset or machine check has been indicated, the contents of the Cerberus status register contains more detailed information on the cause.

Reset

A reset may be caused by a Cerberus reset, a write of the Cerberus control register which sets the **reset** bit, or internally detected errors including meltdown detection, and double machine check.

A reset causes the Euterpe processor to set the configuration to minimum power and low clock speed, note the cause of the reset in the Cerberus status register, stabilize the phase locked loops, set the local TLB to translate all local virtual addresses to equal physical addresses, and initialize a single event thread to begin execution at the start vector address.

Other system state is left undefined by reset and must be explicitly initialized by software; this explicitly includes the main thread state, global TLB state, superspring state, Hermes channel interfaces, Mnemosyne memory and Cerberus interface devices. The code at the start vector address is responsible for initializing these remaining system facilities, and reading further bootstrap code from a series of standard interface devices to be specified.

Power-on Reset

A reset occurs upon initial power-on. The cause of the reset is noted by initializing the Cerberus status register and other registers to the reset values noted below.

Cerberus-grounded Reset

A reset occurs upon observing that the Cerberus SD data signal has been at a logic low level for at least 33 cycles of the Cerberus SC clock signal. The cause of the reset is noted by initializing the Cerberus status register and other registers to the reset values noted below.

MU 0023383

Cerberus Control Register Reset

A reset occurs upon writing a one to the **reset** bit of the Cerberus control register. The cause of the reset is noted by initializing the Cerberus status register and other registers to the reset values noted below.

Meltdown Detected Reset

A reset occurs if the temperature is above the threshold set by the **meltdown margin** field of the Cerberus configuration register. The cause of the reset is noted by setting the **meltdown detected** bit of the Cerberus status register.

Double Machine Check Reset

A reset occurs if a second machine check occurs that prevents recovery from the first machine check. Specifically, the occurrence of an **exception in event thread, watchdog timer error,** or **Cerberus transaction error** while any machine check cause bit is still set in the Cerberus status register results in a double machine check reset. The cause of the reset is noted by setting the **double machine check** bit of the Cerberus status register.

Clear

Writing a one to the **clear** bit of the Cerberus control register invokes a logic clear. A logic clear causes the Euterpe processor to set the configuration to the power and swing levels written in deferred state to the Cerberus power and swing registers, configuration register and Hermes channel configuration registers, stabilize the phase locked loops, set the local TLB to translate all local virtual addresses to equal physical addresses, and initialize a single event thread to begin execution at the start vector address. The cause of the clear is noted by leaving the **clear** bit of the Cerberus control register set to a one (1) at the end of the logic clear.

Machine Check

Detected hardware errors, such as communications errors in one of the Hermes channels or the Cerberus bus, a watchdog timeout error, or internal cache parity errors, invoke a machine check. A machine check will set the local TLB to translate all local virtual addresses to equal physical addresses, note the cause of the exception in the Cerberus status register, and transfer control of the event thread to the start vector address. This action is similar to that of a reset, but differs in that the configuration settings, main thread state, and Cerberus and Mnemosyne state are preserved.

Recovery from machine checks depends on the severity of the error and the potential loss of information as a direct cause of the error. The start vector address is designed to reach instruction memory accessed via Cerberus, so that operation of machine check diagnostic and recovery code need not depend on proper operation or contents of any Hermes channel device. The program counter and register file state of the event thread prior to the machine check is lost (except for the portion of the program counter saved in the Cerberus status register), so diagnostic and recovery code must not assume that the register file state is indicative of the prior operating state of the event thread. The state of the main thread is frozen similarly to that of a main thread exception.

Machine check diagnostic code determines the cause of the machine check from the processor's Cerberus status register, and as required, the Cerberus status and other registers of devices connected to the ByteChannels. Any outstanding memory transactions may be recovered by a combination of software to re-issue outstanding writes, and by aborting and restarting the main thread execution pipeline to purge outstanding reads.

Because Cerberus operates much more slowly than the peak speed of the Euterpe processor under normal operation, machine check diagnostic and recovery code will generally consume enough time that real-time interface performance targets may have been missed. Consequently, the machine check recovery software may need to repair further damage, such as interface buffer underruns and overruns as may have occurred during the intervening time.

This final recovery code, which re-initializes the state of the interface system and recovers a functional event thread state, may return to using the complete machine resources, as the condition which caused the machine check will have been resolved.

The following table lists the causes of machine check errors.

Parity or uncorrectable error in Euterpe cache
Parity or uncorrectable error in Mnemosyne cache
Parity or uncorrectable error in Calliope memory
Parity or uncorrectable error in system-level memory
Communications error in Hermes channels
Communications error in Cerberus bus
Event Thread exception
Watchdog timer

machine check errors

Parity or Uncorrectable Error in Cache

When a parity or uncorrectable error occurs in a Euterpe or Mnemosyne cache, such an error is generally non-recoverable. These errors are non-recoverable because the data in such caches may reside anywhere in memory, and because the data in such caches may be the only up-to-date copy of that memory contents. Consequently, the entire contents of the memory store is lost, and the severity of the error is high enough to consider such a condition to be a system failure.

The machine check provides an opportunity to report such an error before shutting down a system for repairs.

There are specific means by which a system may recover from such an error without failure, such as by restarting from a system-level checkpoint, from which a consistent memory state can be recovered.

Parity or Uncorrectable Error in Memory

When a parity of uncorrectable error occurs in Mnemosyne or Calliope memory, such an error may be partially recoverable. The contents of the affected area of memory is lost, and consequently the tasks associated with that memory must generally be aborted, or resumed from a task-level checkpoint. If the contents of the affected memory can be recovered from mass storage, a complete recovery is possible.

If the affected memory is that of a critical part of the operating system, such a condition is considered a system failure, unless recovery can be accomplished from a system-level checkpoint.

Communications Error in Hermes Channels

A communications error in Hermes channels, such as a check byte error, command error, or timeout error, is generally fully recoverable.

Bits corresponding to the affected Hermes channel are set in the processor's Cerberus status register. Recovery software should determine which devices are affected, by querying the Cerberus status register of each device on the affected Hermes channels.

Read and write transactions may have been underway at the time of a machine check. Because the machine check freezes the Hermes channel(s), these transactions will not be completed. Recovery software must search through the memory interface buffers for uncompleted write operations and re-issue them as stores, then must reset the memory interface buffers to a known state.

Communications Error in Cerberus Bus

A communications error in the Cerberus bus, such as a Cerberus transaction error is generally fully recoverable. A Cerberus transaction error (due to timeout) may result from normal system self-configuration operations to determine the existence of optional devices in the system.

Watchdog Timeout Error

A watchdog timeout error indicates a general software or hardware failure. Such an error is generally treated as non-recoverable and fatal.

Event Thread Exception

When an event thread suffers an exception, the cause of the exception and a portion of the virtual address at which the exception occurred are noted in the Cerberus status register. Because under normal circumstances, the event thread should be designed not to encounter exceptions, such exceptions are treated as non-recoverable, fatal errors.

Start Vector Address

The start vector address is used to initialize the event thread with a program counter upon a reset, clear or machine check. These causes of such initialization can be differentiated by the contents of the Cerberus status register.

The start vector address is a virtual address which, when "translated" by the local TLB to a physical address, is designed to access node number zero on the local Cerberus network, which will ordinarity contain an interface to the bootstrap ROM code. The Cerberus/Bootstrap ROM space is chosen to minimize the number of internal Terpsichore resources and Terpsichore interfaces that must be operated to begin execution or recover from a machine check.

virtual address	description
0x0003 0000 0000 0000	start vector address

Bootstrap Code

Bootstrap code requirements are a necessary part of the Terpsichore System Architecture, but remains to be specified in a later version of this document.

The basic requirements of Terpsichore bootstrap code include power-on initialization of Euterpe, Calliope, and Mnemosyne devices, using Cerberus control registers; handling of machine checks, selection of an interface from which further bootstrap code is obtained. Interfaces should be scanned in a priority-based ordering which gives highest priority to removable/replaceable read-only storage devices, then removable/replaceable read-write devices, then network interfaces, then non-removable storage devices.

Cerberus Registers

Cerberus registers are internal read/only and read/write registers which provide an implementation-independent mechanism to query and control the configuration of devices in a Terpsichore system. By the use of these registers, a user of a Terpsichore system may tailor the use of the facilities in a general-purpose implementation for maximum performance and utility. Conversely, a supplier of a Terpsichore system component may modify facilities in the device without compromising compatibility with earlier implementations. These registers are accessed via the Cerberus serial bus.

As a device component of a Terpsichore system, each Euterpe processor contains a set of Cerberus-accessable configuration registers. Additional sets of configuration registers are present for each additional device in a Euterpe system, including Mnemosyne Memory devices, and Calliope interface devices.

Read/only registers supply information about the Terpsichore system implementation in a standard, implementation-independent fashion. Terpsichore software may take advantage of this information, either to verify that a compatible

implementation of Mnemosyne is installed, or to tailor the use of the part to conform to the characteristics of the implementation.

The read/only registers occupy addresses 0..5. An attempt to write these registers may cause a normal or an error response.

Read/write registers select operating modes and select power and voltage levels for gates and signals. The read/write registers occupy addresses 6..9 and 25..43.

Reserved registers in the range 10..24 and 44..63 must appear to be read/only registers with a zero value. An attempt to write these registers may cause a normal or an error response.

Reserved registers in the range 64..2^{16}-1 may be implemented either as read/only registers with a zero value, or as addresses which cause an error response if reads or writes are attempted.

The format of the registers is described in the table below. The **octlet** is the Cerberus address of the register, bits indicate the position of the field in a register. The **value** indicated is the hard-wired value in the register for a read/only register, and is the value to which the register is initialized upon a reset for a read/write register. If a reset does not initialize the field to a value, or if initialization is not required by this specification, a ˢ is placed in or appended to the value field. The **range** is the set of legal values to which a read/write register may be set. The **interpretation** is a brief description of the meaning or utility of the register field; a more comprehensive description follows this table.

octlet	bits	field name	value	range	interpretation
0	63..16	**architecture code**	0x00 40 a3 24 69 93		Identifies processor device as compliant with MicroUnity Euterpe processor architecture.
	15..0	**architecture revision**	0x01 00		Device complies with architecture version 1.0.

octlet	bits	field name	value	range	interpretation
1	63..16	**implementor code**	0x00 40 a3 d2 b6 7f		Identifies Euterpe processor device as implemented by MicroUnity.
	15..0	**implementor revision**	0x01 00		Implementation version 1.0.

octlet	bits	field name	value range	interpretation
2	63..16	**manufacturer code**	0x00 40 a3 69 db 3f	Identifies initial manufacturer of Euterpe processor device implemented by MicroUnity.
	15..0	**manufacturer revision**	0x01 00	Manufacturing version 1.0.

octlet	bits	field name	value range	interpretation
3	63..16	**serial number**	0	This device has no serial number capability.
		dynamic address	0	This device has no dynamic addressing capability.

octlet	bits	field name	value range	interpretation
4	63..60	**A**	4	Size of a Hermes address
	59..56	**$\log_2 W$**	3	Size of a Hermes word
	55..0		0	Reserved for definition in later revision of Euterpe architecture

octlet	bits	field name	value range	interpretation
5	63.0		0	Reserved for definition in later revision of Euterpe architecture

octlet	bits	field name	value	range	interpretation
6	63	**reset**	1	0..1	set to invoke device's circuit reset
	62	**clear**	1	0..1	set to invoke device's logic clear
	61	**selftest**	0	0..1	set to invoke device's selftest: bits 60..48 may indicate depth of selftest
	60	**defer writes**	0*	0..1	set to cause writes to octlets 25..43 to be deferred until the next logic-clear or non-deferred write.
	59..48	**0**	0	0	Reserved
	47..44	**Hermes channel expansion**	0	0	Reserved for additional Hermes channel disable bits.
	43..32	**Hermes channel disable**	4095	0..4095	For each Hermes channel, set to cause input channel to be ignored and idles to be generated. Upon clearing the bit, the input channel phase adjustment is reset, and after a suitable delay, the input and Hermes output channel links are available for use.
	31..20	**0**	0	0	Reserved
	19..16	**channel under test**	0*	0..11	Channel on which cidle 0 and cidle 1 are transmitted in place of normal idle pattern (0, 255), and from which raw input bytes are sampled.
	15..8	**cidle_0**	0*	0..255	Value transmitted on idle Hermes output channel when output clock zero (0).
	7..0	**cidle_1**	255*	0..255	Value transmitted on idle Hermes output channel when output clock one (1).

octlet	bits	field name	value	range	interpretation
7	63	**reset/clear/ selftest complete**	1	0..1	This bit is set when a reset, clear or selftest operation has been completed.
	62	**reset/clear/ selftest status**	1	0..1	This bit is set when a reset, clear or selftest operation has been completed successfully.
	61	**meltdown detected**	0	0..1	This bit is set when the meltdown detector has caused a reset.
	60	**double machine check**	0	0..1	This bit is set when a double machine check has caused a reset.
	59	**other reset cause**	0	0..1	This bit is reserved for indicating additional causes of reset.
	58	**exception in event thread**	0	0..1	This bit is set when an exception in event thread has caused a machine check.
	57	**watchdog timeout error**	0	0..1	This bit is set when a watchdog timeout has caused a machine check.
	56	**Cerberus transaction error**	0	0..1	This bit is set when a Cerberus transaction error has caused a machine check.
	55	**Hermes channel check byte error**	0	0..1	This bit is set when a Hermes channel check byte error has caused a machine check.
	54	**Hermes channel command error**	0	0..1	This bit is set when a Hermes channel command error has caused a machine check.
	53	**Hermes channel timeout error**	0	0..1	This bit is set when a Hermes channel timeout has caused a machine check.
	52..48	**0**	0*	0	Reserved for other machine check causes.
	47..32	**machine check detail**	0*	0..40 95	Set to indicate exception code if Exception in event thread. Set to bitmap of which Hermes channels if Hermes channel error.
	31..16	**machine check program counter**	0	0	Set to indicate bits 31..16 of the value of the event thread program counter at the initiation of a machine check.
	15..8	**raw 0**	*	0..25 5	Value sampled on specified Hermes channel when input clock is zero (0).

7..0	**raw 1**	*	0..255	Value sampled on specified Hermes channel immediately following sample value in **raw 0** register.

octlet	bits	field name	value range		interpretation
8	63..0	**indirect address**	0*	0..264-1	Write to this register to set physical address used for reads and writes to indirect data register.

octlet	bits	field name	value value		interpretation
9	63..0	**indirect data**	*	0..264-1	Read and write to this register to reach physical addresses not otherwise accessable via Cerberus.

octlet	bits	field name	value range		interpretation
10..24	63..0	**0**	0	0	Reserved for expansion of Cerberus registers upward or knobcity registers downward.

octlet	bits	field name	value range		interpretation
25	63..56	**Unassigned Custom knob**	121	1..127	Knob settings for Unassigned custom circuits.
	55..48	**Unassigned Custom knob**	121	1..127	Knob settings for Unassigned custom circuits.
	47..40	**CI Tag knob**	121	1..127	Knob settings for CI Tag circuits.
	39..32	**CD Tag knob**	121	1..127	Knob settings for CD Tag circuits.
	31..24	**TLB knob**	121	1..127	Knob settings for TLB circuits.
	23..16	**Branch Target Cache knob**	121	1..127	Knob settings for Branch Target Cache circuits.
	15..8	**I Cache knob**	121	1..127	Knob settings for Instruction Cache circuits.
	7..0	**Eastside Repeater knob**	121	1..127	Knob settings for Eastside Repeater circuits.

MU 0023393

octlet	bits	field name	value range		interpretation
26	63..56	**spar 1,2 knob**	121	1..127	Knob settings for SOFA region spar 1, 2.
	55..48	**spar 1,2 knob**	121	1..127	Knob settings for SOFA region spar 1, 2.
	47..40	**spar 1,2 knob**	121	1..127	Knob settings for SOFA region spar 1, 2.
	39..32	**spar 1,2 knob**	121	1..127	Knob settings for SOFA region spar 1, 2.
	31..24	**spar 0 knob**	121	1..127	Knob settings for SOFA region spar 0.
	23..16	**spar 0 knob**	121	1..127	Knob settings for SOFA region spar 0.
	15..8	**spar 0 knob**	121	1..127	Knob settings for SOFA region spar 0.
	7..0	**spar 0 knob**	121	1..127	Knob settings for SOFA region spar 0.

octlet	bits	field name	value range		interpretation
27	63..56	**spar 5,6 knob**	121	1..127	Knob settings for SOFA region spar 5,6.
	55..48	**spar 5,6 knob**	121	1..127	Knob settings for SOFA region spar 5,6.
	47..40	**spar 5,6 knob**	121	1..127	Knob settings for SOFA region spar 5,6.
	39..32	**spar 5,6 knob**	121	1..127	Knob settings for SOFA region spar 5,6.

31..24	**spar 3,4 knob**	121	1..127	Knob settings for SOFA region spar 3,4.
23..16	**spar 3,4 knob**	121	1..127	Knob settings for SOFA region spar 3,4.
15..8	**spar 3,4 knob**	121	1..127	Knob settings for SOFA region spar 3,4.
7..0	**spar 3,4 knob**	121	1..127	Knob settings for SOFA region spar 3,4.

octlet	bits	field name	value range		interpretation
28	63..56	**spar 9,10 knob**	121	1..127	Knob settings for SOFA region spar 9,10.
	55..48	**spar 9,10 knob**	121	1..127	Knob settings for SOFA region spar 9,10.
	47..40	**spar 7,8 knob**	121	1..127	Knob settings for SOFA region spar 7,8.
	39..32	**spar 7,8 knob**	121	1..127	Knob settings for SOFA region spar 7,8.
	31..24	**spar 7,8 knob**	121	1..127	Knob settings for SOFA region spar 7,8.
	23..16	**spar 7,8 knob**	121	1..127	Knob settings for SOFA region spar 7,8.
	15..8	**Clocks knob**	121	1..127	Knob settings for clock circuits.
	7..0	**PLL knob**	85	1..127	Knob settings for PLL circuits.

octlet	bits	field name	value range		interpretation
29	63..56	**spar 13,14 knob**	121	1..127	Knob settings for SOFA region spar 13,14.
	55..48	**spar 13,14 knob**	121	1..127	Knob settings for SOFA region spar 13,14.
	47..40	**spar 11,12 knob**	121	1..127	Knob settings for SOFA region spar 11,12.
	39..32	**spar 11,12 knob**	121	1..127	Knob settings for SOFA region spar 11,12.
	31..24	**spar 11,12 knob**	121	1..127	Knob settings for SOFA region spar 11,12.
	23..16	**spar 11,12 knob**	121	1..127	Knob settings for SOFA region spar 11,12.
	15..8	**spar 9,10 knob**	121	1..127	Knob settings for SOFA region spar 9,10.
	7..0	**spar 9,10 knob**	121	1..127	Knob settings for SOFA region spar 9,10.

octlet bits field name value range interpretation

30	63..56	**Hermes channel knob**	121	1..127	Knob settings for Hermes channel circuits.
	55..48	**Westside Repeaters knob**	121	1..127	Knob settings for Westside Repeater circuits.
	47..40	**D Cache knob**	121	1..127	Knob settings for Data Cache circuits.
	39..32	**Spring knob**	121	1..127	Knob settings for Spring circuits.
	31..24	**Unassigned Custom knob**	121	1..127	Knob settings for Unassigned custom circuits.
	23..16	**Unassigned Custom knob**	121	1..127	Knob settings for Unassigned custom circuits.
	15..8	**spar 13,14 knob**	121	1..127	Knob settings for SOFA region spar 13,14.
	7..0	**spar 13,14 knob**	121	1..127	Knob settings for SOFA region spar 13,14.

octlet	bits	field name	value	range	interpretation
31	63	**0**	0	0	Reserved
	62..58	**resistor fine tuning**	20*	0..31	Set to fine tune resistor termination value
	57..56	**swing fine tuning**	1*	0..3	Set to fine-tune voltage swing and reference level knob settings.
	55	**0**	0	0	Reserved
	54..52	**process control**	5	4..6	Set based on value read from PMOS drive strength, used to fine-tune resistor values in knob settings.
	51	**0**	0	0	Reserved
	50..48	**PMOS drive strength**	*	0..7	This read/only field indicates the drive strength of PMOS devices expressed as a digital binary value.
	47..43	**PLL1 divide ratio**	8*	8..23	PLL1 divider ratio
	42	**PLL1 feedback bypass**	1*	0..1	Set to invoke PLL1 feedback bypass.
	41	**PLL1 range**	0*	0..1	Set for operation at high frequency (above 0.xxx GHz); cleared for operation at low frequency (below 0.yyy GHz).
	40	**PLL prescaler bypass**	0	0..1	Set to invoke PLL0 and PLL1 prescaler bypass; otherwise divide input clock by 10.
	39..35	**PLL0 divide ratio**	8*	8..23	PLL0 divider ratio
	34	**PLL0 feedback bypass**	1	0..1	Set to invoke PLL0 feedback bypass.
	33	**PLL0 range**	0	0..1	Set for operation at high frequency (above 0.xxx GHz); cleared for operation at low frequency (below 0.yyy GHz).
	32	**conversion prescaler bypass**	0	0..1	Set to invoke temperature conversion prescaler bypass, otherwise divide input clock by 10.
	31..24	**analog measurement**	0	0..255	Set to measure analog levels at various test points within device.
	23..22	**meltdown threshold**	0	0..3	Set to perform margin testing of the meltdown detector.
	21	**conversion start**	0*	0..1	Setting this bit causes the conversion to begin. The bit remains set until conversion is complete
	20	**0**	0	0	Reserved. (selection extension)

19..16	**conversion selection**	0*	0..9	Field selects which of ten measurements are taken
15..10	**0**	0	0	Reserved. (counter extension)
9..0	**conversion counter**	0*	0..1023	This field is set to the two's complement of the downslope count. The counter counts upward to zero, and then continues counting on the upslope until conversion completes.

octlet	bits	field name	value range		interpretation
32..43	63	**0**	0	0	Reserved
	62	**quadrature bypass**	0*	0..1	Setting this bit causes the quadrature circuit to be bypassed; the input clock signal is used directly.
	61	**quadrature range**	0*	0..1	Set to 0 if the Hermes channel is operating at a low frequency; 1 if the Hermes channel is operating at a high frequency.
	60	**output termination**	1	0..1	Set to enable output terminators. Cleared to disable output terminators.
	59..57	**termination resistance**	1	0..7	Set termination resistance level.
	56..54	**output current**	1	0..7	Set output current level.
	53..48	**skew bit 7**	1	0..63	Set delay in Ho7 skew circuit.
	47..42	**skew bit 6**	1	0..63	Set delay in Ho6 skew circuit.
	41..36	**skew bit 5**	1	0..63	Set delay in Ho5 skew circuit.
	35..30	**skew bit 4**	1	0..63	Set delay in Ho4 skew circuit.
	29..24	**skew bit 3**	1	0..63	Set delay in Ho3 skew circuit.
	23..18	**skew bit 2**	1	0..63	Set delay in Ho2 skew circuit.
	17..12	**skew bit 1**	1	0..63	Set delay in Ho1 skew circuit.
	11..6	**skew bit 0**	1	0..63	Set delay in Ho0 skew circuit.
	5..0	**skew clk**	1	0..63	Set delay in HoC skew circuit.

octlet	bits	field name	value range		interpretation
44..63	63..0	**0**	0	0	Reserved for use with additional Hermes channel interfaces

octlet	bits	field name	value range		interpretation
64.. 65536	63..0	**0**	0	0	Reserved for use with later revisions of the architecture.

configuration memory space

Identification Registers

The identification registers in octlets 0..3 comply with the requirements of the Cerberus architecture.

MicroUnity's company identifier is: 0000 0000 0000 0010 1100 0101.

MicroUnity's architecture code for Euterpe is specified by the following table:

Internal code name	Code number
Euterpe	0x00 40 a3 24 69 93

Euterpe architecture revisions are specified by the following table:

Internal code name	Code number
1.0	0x01 00

MicroUnity's Euterpe implementor codes are specified by the following table:

Internal code name	Code number
MicroUnity	0x00 40 a3 d2 56 7f

MicroUnity's Euterpe, as implemented by MicroUnity, uses implementation codes as specified by the following table:

Internal code name	Revision number
1.0	0x01 00

MicroUnity's Euterpe, as implemented by MicroUnity, uses manufacturer codes as specified by the following table:

Internal code name	Code number
MicroUnity	0x00 40 a3 69 db 3f

MicroUnity's Euterpe, as implemented by MicroUnity, and manufactured by MicroUnity, uses manufacturer revisions as specified by the following table:

Internal code name	Code number
1.0	0x01 00

MU 0023398

Architecture Description Registers

The architecture description registers in octlets 4 and 5 comply with the Cerberus specification and contain a machine-readable version of the architecture parameters: A and W described in this document.

These registers are still under construction and will contain non-zero values in a later revision of this document.

Parameters will describe number of Hermes ports, size of internal caches, integrat ion of Call iope and Mnemosyne functions.

Control Register

The control register is a 64-bit register with both read and write access. It is altered only by Cerberus accesses; Euterpe does not alter the values written to this register.

The **reset** bit of the control register complies with the Cerberus specification and provides the ability to reset an individual Euterpe device in a Terpsichore system. Writing a one (1) to this bit is equivalent to a power-on reset or a broadcast Cerberus reset (low level on SD for 33 cycles) and resets configuration registers to their power-on values, which is an operating state that consumes minimal current, and also causes all internal high-bandwidth logic to be reset. The duration of the reset is sufficient for the operating state changes to have taken effect. At the completion of the reset operation, the **reset/clear/selftest complete** bit of the status register is set, the **reset/clear/selftest status** bit of the status register is set, and the **reset** bit of the control register is set.

The **clear** bit of the control register complies with the Cerberus specification and provides the ability to clear the logic of an individual Euterpe device in a system. Writing a one (1) to this bit causes all internal high-bandwidth logic to be reset, as is required after reconfiguring power and swing levels. The duration of the reset is sufficient for any operating state changes to have taken effect. At the completion of the reset operation, the **reset/clear/selftest complete** bit of the status register is set, the **reset/clear/selftest status** bit of the status register is set, and the **clear** bit of the control register is set.

The **selftest** bit of the control register complies with the Cerberus specification and provides the ability to invoke a selftest on an individual Euterpe device in a system. However, Euterpe does not define a selftest mechanism at this time, so setting this bit will immediately set the **reset/clear/selftest complete** bit and the **reset/clear/selftest status** bit of the status register.

The **channel under test** field of the control register provides a mechanism to test and adjust skews on a single Hermes channel at a time. The field is set to the channel number for which the **cidle 0, cidle 1, raw 0,** and **raw 1** fields are active.

The **cidle 0** and **cidle 1** fields of the control register provide a mechanism to repeatedly sent simple patterns on the selected Hermes output channel for purposes of testing and skew adjustment. For normal operation, the **cidle 0** field must be set to zero (0), and the **cidle 1** field must be set to all ones (255).

Status Register

The status register is a 64-bit register with both read and write access, though the only legal value which may be written is a zero, to clear the register. The result of writing a non-zero value is not specified.

The **reset/clear/selftest complete** bit of the status register complies with the Cerberus specification and is set upon the completion of a reset, clear or selftest operation as described above.

The **reset/clear/selftest status** bit of the status register complies with the Cerberus specification and is set upon the successful completion of a reset, clear or selftest operation as described above.

The **meltdown detected** bit of the status register is set when the meltdown detector has discovered an on-chip temperature above the threshold set by the **meltdown threshold** field of the Cerberus configuration register, which causes a reset to occur.

The **double machine check** bit of the status register is set when a second machine check occurs that prevents recovery from the first machine check, or which is indicative of machine check recovery software failure. Specifically, the occurrence of an **exception in event thread**, **watchdog timer error**, or **Cerberus transaction error** while any machine check cause bit of the status register is still set, or any Hermes error while the **exception in event thread** bit of the status register is set in the Cerberus status register results in a double machine check reset.

The **other reset** cause bit of the status register is reserved for the indication of other causes of reset.

The **exception in event thread** bit of the status register is set when an event thread suffers and exception, which causes a machine check. The exception code is loaded into the **machine check detail** field of the status register.

The **watchdog timeout error** bit of the status register is set when the **watchdog timer** register is equal to the **clock cycle** register, causing a machine check.

The **Cerberus transaction error** bit of the status register is set when a Cerberus transaction error (bus timeout, invalid transaction code, invalid address) has caused a machine check. Note that Cerberus aborts, including locally detected parity errors, should cause bus retries, not a machine check.

The **Hermes check byte error** bit of the status register is set when a Hermes check byte error has caused a machine check. The bit corresponding to the Hermes channel number which has suffered the error is set in the **machine check detail field** of the status register.

The **Hermes command error** bit of the status register is set when a Hermes command error has caused a machine check. The bit corresponding to the

Hermes channel number which has suffered the error is set in the **machine check detail field** of the status register.

The **Hermes timeout error** bit of the status register is set when aByteChannel timeout error has caused a machine check. The bit corresponding to the Hermes channel number which has suffered the error is set in the **machine check detail field** of the status register.

The **machine check detail** field of the status register is set when a machine check has been completed. For a Hermes channel error (check byte, command or timeout), the value indicates, via a bit-mask, ByteChannels for which machine checks have been reported. For an exception in event thread, the value indicates the type of exception for which the most recent machine check has been reported.

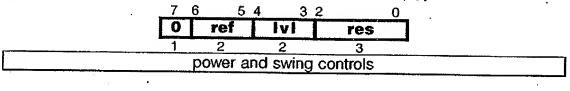
The **machine check program counter** field of the status register is loaded with bits 31..16 of the event thread program counter at which the most recent machine check has occurred. The value in this field provides a limited diagnostic capability for purposes of software development, or possibly for error recovery.

The **raw 0** and **raw 1** fields of the status register contain the values obtained from two adjacent samples of the specified Hermes input channel. The **raw 0** field contains a value obtained when the input clock was zero (0), and the **raw 1** field contains the value obtained on the immediately following sample, when the input clock was (1). Euterpe must ensure that reading the status register produces two adjacent samples regardless of the timing of the status register read operation on Cerberus. These fields are read for purposes of testing and control of skew in the Hermes channel interfaces.

Power and Swing Calibration Registers

Euterpe uses a set of configuration registers to control the power and voltage levels used for internal high-bandwidth logic and memory. The details of programming these registers are described below.

Eight-bit fields separately control the power and voltage levels used in a portion of the Euterpe circuitry. Each such field contains configuration data in the following format:



The range of valid values and the interpretation of the fields is given by the following table:

field	value	interpretation
0	0	Reserved
ref	0..3	Set reference voltage level
lvl	0..3	Set voltage swing level.
res	0..7	Set resistor load value.

Power and swing control field interpretation

The reference voltage level, voltage swing level and resistor load value are model figures for a full-swing, lowest-power logic gate output. The actual voltage levels and resistor load values used in various circuits is geometrically related to the values in the tables below. Designed typical, full-speed settings for the ref, lvl and res fields are ref=250 millivolts, lvl=500 millivolts, and res=2.5 kilohms.

The ref field, together with the swing fine tuning field of the configuration register, control the reference voltage level used for logic circuits in the specified knob domain. Values and interpretations of the ref field are given by the following table, with units in millivolts:

	swing fine tuning			
ref	0	1	2	3
0	138	150	163	175
1	188	200	213	225
2	238	250	263	275
3	288	300	325	350

Reference level control field interpretation

The lvl field, together with the swing fine tuning field of the configuration register, control the voltage swing level used for logic circuits in the specified knob domain. Values and interpretations of the lvl field are given by the following table, with units in millivolts:

	swing fine tuning			
lvl	0	1	2	3
0	275	300	325	350
1	375	400	425	450
2	475	500	525	550
3	575	600	650	700

Voltage swing level control field interpretation

The **res** field, together with the **process control** field of the configuration register, control the PMOS load resistance value used for logic circuits in the specified knob domain. Values and interpretations of the **lvl** field are given by the following table, with units in kilohms. The table below gives resistance values with nominal process parameters.

	process control							
res	0	1	2	3	4	5	6	7
0	undefined							
1		2.5	5.0	7.5	10.	13.	15.	18.
2		1.3	2.5	3.8	5.0	6.3	7.5	8.8
3		.83	1.7	2.5	3.3	4.2	5	5.8
4		.63	1.3	1.9	2.5	3.1	3.8	4.4
5		.50	1.0	1.5	2.0	2.5	3	3.5
6		.42	.83	1.3	1.7	2.1	2.5	2.9
7		.36	.71	1.1	1.4	1.8	2.1	2.5

Resistor control field interpretation

When the **process control** field of the configuration register is set equal to the PMOS drive strength field of the configuration register, nominal PMOS load resistance values are as given by the following table, with units in kilohms.

res	PMOS load resistance
0	undefined
1	13.
2	6.3
3	4.2
4	3.1
5	2.5
6	2.1
7	1.8

When Mnemosyne is reset, a default value of 0 is loaded into each **0** field, 3 in each **ref** field, 3 in each **lvl** field and 1 in each **res** field, which is a byte value of 121. The **process control** field of the configuration register is set to 5, and the **swing fine tuning** field is set to 1. These settings correspond to a chip with nominal processing parameters, low power and high voltage swing operation.

For nominal operating conditions, the **ref** field is set to 2, the **lvl** field is set to 2, and the **res** field is set to 5, which is byte value of 85. The **process control** field is set equal to the **PMOS strength** field, and the **swing fine tuning** field is set to 1.

MU 0023403

Configuration Register

A Configuration register is provided on the Euterpe processor to control the fine-tuning of the Hermes channel configuration, to control the global process

parameter settings, to control the two phase-locked loop frequency generators, and to control the temperature sensors and read temperature values.

The **resistor fine tuning** field of the configuration register controls the analog bias settings for PMOS loads in Hermes channel input and output termination circuits, in order to accomodate variations in circuit paramaters due to the manufacturing process, and to provide fine-tuning of the input and output impedence levels. Under normal operating conditions, four times (4*) the value read from the **PMOS drive strength** field should be written into the **resistor fine tuning** field. In order to provide fine-tuning of the input and output impedence levels, an external measurement of the impedence or voltage levels is required. An change of the **resistor fine tuning** field causes a proportional change in the input and output impedence levels. The interpretation of the field is given by the table:

value	resistor fine tuning
0..13	Reserved
14..19	increase PMOS conductance to nominal*20/value.
20	use PMOS loads at nominal conductance.
21..31	decrease PMOS conductance to nominal*20/value.

The **swing fine tuning** field of the configuration register controls a small offset in the reference voltage and logic swing voltages for internal logic circuits. The swing fine tuning voltage is added to the output current field of the Hermes channel configuration registers to select the output current. The interpretation of the field is given by the table.

value	swing fine tuning	reference fine tuning
0	-25 mv	-12 mv
1	0	0
2	+25 mv	+13 mv
3	+50 mv or +100 mv	+25 mv or +50 mv

swing_fine_tuning field interpretation

The **process control** field of the configuration register controls the analog bias settings for PMOS loads in internal logic circuits, in order to accomodate variations in circuit parameters due to the manufacturing process. Under normal operating conditions, the value read from the **PMOS drive strength** field should be written into the **process control** field. The interpretation of the field is given by the table:

value	process control
0	Reserved
1	increase PMOS conductance to 5.00*nominal.
2	increase PMOS conductance to 2.50*nominal.
3	increase PMOS conductance to 1.66*nominal.
4	increase PMOS conductance to 1.25*nominal.
5	use PMOS loadsat nominal conductance.
6	decrease PMOS conductance to 0.83*nominal.
7	decrease PMOS conductance to 0.71*nominal.

The **PMOS drive strength** field of the configuration register is a read/only field that indicates the drive strength, or conductance gain, of PMOS devices on the Euterpe chip, expressed as a digital binary value. This field is used to calibrate the power and voltage level configuration, given variations in process characteristics of individual devices. The interpretation of the field is given by the table:

value	PMOS drive strength
0	Reserved
1	0.2*nominal
2	0.4*nominal
3	0.6*nominal
4	0.8*nominal
5	nominal
6	1.2*nominal
7	1.4*nominal

There are two identical phase locked-loop (PLL) frequency generators, designated PLL0 and PLL1. These PLLs generate internal and external clock signals of configurable frequency, based upon an input clock reference of either 50 MHz or 500 MHz. PLL0 controls the internal operating frequency of the Euterpe processor, while PLL1 controls the operating frequency of the Hermes channel interfaces. The configuration fields for PLL0 and PLL1 have identical meanings, described below:

The **PLL0 divide ratio** and **PLL1 divide ratio** fields select the divider ratio for each PLL, where legal values are in the range 8..23. These divider ratios permit clock signals to be generated in the range from 400 MHz to 1.15 GHz, when the input clock reference is at 50 MHz, with prescaling bypassed, or at 500 MHz with prescaling used.

Setting the **PLL0 feedback bypass** bit or the **PLL1 feedback bypass** bit of the configuration register causes the generated clock bypass the PLL oscillator and to operate off the input clock directly. Setting these bits causes the frequency generated to be the optionally prescaled reference clock. These bits are cleared during normal operation, and set by a reset.

The **PLL0 range** field and the **PLL0 range** field of the configuration register are used to select an operating range for the internal PLLs. If the PLL range is set to zero, the PLL will operate at a low frequency (below 0.xxx GHz), if the PLL range is set to one, the PLL will operate at a high frequency (above 0.xxx GHz). At reset this bit is cleared, as the input clock frequency is unknown.

Setting the **PLL prescaler bypass** bit of the configuration register causes the phase-locked loops PLL0 and PLL1 to use the input clock directly as a reference clock. This bit is cleared during normal operation with a 500 MHz input clock, in which the input clock is divided by 10, and is set during normal operation with a 50 MHz input clock. At reset this bit is cleared, as the input clock frequency is unknown.

Setting the **conversion prescaler bypass** bit of the configuration register causes the temperature conversion unit to use the input clock directly as a reference clock. Otherwise, clearing this bit causes the input clock to be divided by 10 before use as a reference clock. The reference clock frequency of the temperature conversion unit is nominally 50 MHz, and in normal operation, this bit should be set or cleared, depending on the input clock frequency. At reset this bit is cleared, as the input clock frequency is unknown.

The **meltdown margin** field controls the setting of the threshold at which meltdown is signalled. This field is used to test the meldown prevention logic. The interpretation of the field is given by the table below with a tolerance of ±6 degrees C, and 5 degrees C hysteresis:

value	meltdown threshold
0	150 degrees C
1	90 degrees C
2	50 degrees C
3	20 degrees C

The **conversion start** bit controls the initiation of the conversion of a temperature sensor or reference to a digital value. Setting this bit causes the conversion to begin, and the bit remains set until conversion is complete, at which time the bit is cleared.

The **conversion selection** field controls which sensor or reference value is converted to a digital value. The interpretation of the field is given by the table below:

value	conversion selected
0	local temperature sensor
1	local temperature reference
2	remote 0 temperature sensor
3	remote 0 temperature reference
4	remote 1 temperature sensor
5	remote 1 temperature reference
6	remote 2 temperature sensor
7	remote 2 temperature reference
8	remote 3 temperature sensor
9	remote 3 temperature reference
10..15	Reserved

MU 0023407

The **conversion counter** field is set to the two's complement of the downslope count. The counter counts upward to zero, at which point the upslope ramp begins, and continues counting on the upslope until the conversion completes.

Hermes channel Configuration Registers

Configuration registers are provided on the Interpe processor to control the timing, current levels, and termination resistance for each of the twelve Hermes channel high-bandwidth channels. A configuration register is dedicated to the control of each Hermes channel, and additional information in the configuration register at octlet 31 controls aspects of the Hermes channel circuits in common. The Hermes channel configuration registers are Cerberus registers 32..43, where 32 corresponds to Hermes channel 0, and where 43 corresponds to Hermes channel 11.

The **quadrature bypass** bit controls whether the HiC clock signal is delayed by approximately ¼ of a HiC clock cycle to latch the $Hi_{7..0}$ bits. In normal, full speed operation, this bit should be cleared to a zero value. If this bit is set, the quadrature delay is defeated and the HiC clock signal is used directly to latch the $Hi_{7..0}$ bits.

The **quadrature range** bit is used to select an operating range to the quadrature delay circuit. If the quadrature range is set to zero, the circuit will operate at a low frequency (below 0.xxx GHz), if the quadrature range is set to one, the circuit will operate at a high frequency (above 0.xxx GHz).

The **output termination** bit is used to select whether the output circuits are resistively terminated. If the bit is set to a zero, the output has high impedence; if the bit is set to one, the output is terminated with a resistance equal to the input termination. At reset, this bit is set to one, terminating the output.

The **termination resistance** field is used to select the impedence at which the Hermes channel inputs, and optionally the Hermes channel outputs are terminated. The resistance level is controlled relative to the setting of the **resistor**

fine tuning field of the configuration register. The interpretation of the field is given by the table, with units in Ohms and nominal PMOS conductance and bias settings:

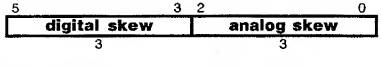
value	termination resistance
0	Reserved
1	250. Ohms
2	125. Ohms
3	83.3 Ohms
4	62.5 Ohms
5	50.0 Ohms
6	41.7 Ohms
7	35.7 Ohms

The **output current** field is used to select the current at which the Hermes channel outputs are operated. The interpretation of the field is given by the table, with units in mA:

value	output current
0	Reserved
1	2. mA
2	4. mA
3	6. mA
4	8. mA
5	10. mA
6	12. mA
7	14. mA

The output voltage swing is the product of the composite termination resistance: (input termination resistance^{-1}+output termination resistance^{-1})$^{-1}$, and the output current. The output voltage swing should be set at or below 700 mV, and is normally set to the lowest value which permits a sufficiently low bit error rate, which depends upon the noise level in the system environment.

The **skew** fields individually control the delay between the internal Hermes channel output clock and each of the HoC and Ho7..0 high bandwidth output channel signals. Each skew field contains two three-bit values, named digital skew and analog skew as shown below:



MU 0023408

The **digital skew** fields set the number of delay stages inserted in the output path of the HoC and the Ho7..0 high-bandwidth output channel signals. The **analog skew** fields control the power level, and thereby control the switching delay, of a single delay stage. Setting these fields permits a fine level of control over the relative skew between output channel signals. Nominal values for the output delay

for various values of the digital skew and analog skew fields are given below, assuming a nominal setting for the **Hermes channel knob**:

digital skew	delay (ps)	plus analog skew
0	0	no
1	320	yes
2	400	yes
3	470	yes
4	570	yes
5	670	yes
6	770	yes
7	870	yes

analog skew	delay (ps)
0	Reserved
1	???
2	???
3	+40
4	+20
5	0
6	-10
7	-20

When Euterpe is reset, a default value of 0 is loaded into the **digital skew** and 1 is loaded into the **analog skew** fields, setting a minimum output delay for the HoC and Ho7..0 signals.

Mnemosyne Memory

MicroUnity's Mnemosyne memory architecture is designed for ultra-high bandwidth systems. The architecture integrates fast communication channels with SRAM caches and interfaces to standard DRAM.

The Mnemosyne interfaces include byte-wide input and output channels intended to operate at rates of at least 1 GHz. These channels provide a packet communication link to synchronous SRAM cache on chip and a controller for external banks of conventional DRAM components. Mnemosyne provides second-level cache and main memory for MicroUnity's Terpsichore system architecture. However, Mnemosyne is useful in many memory applications.

Mnemosyne's interface protocol embeds read and write operations to a single memory space into packets containing command, address, data, and acknowledgement. The packets include check codes that will detect single-bit transmission errors and multiple-bit errors with high probability. As many as eight operations in each device may be in progress at a time. As many as four Mnemosyne devices may be cascaded to expand the cache and memory and to improve the bandwidth of the DRAM memory.

Mnemosyne's SRAM arrays are organized as a set of small blocks, which are combined to provide a cache containing logical memory data of a fixed word size. Dynamically-configured block-level redundancy supports the elimination of faulty blocks without requiring the use of non-volatile or one-time-programmable storage.

Mnemosyne's DRAM interface provides for the direct connection of multiple banks of standard DRAM components to a Mnemosyne device. Variations in access time, size, and number of installed parts all may be accommodated by reading and writing of configuration registers. The interface supports interleaving to enhance bandwidth, and page-mode accesses to improve latency for localized addressing.

Enterpé uses Mnemosyne devices as a second-level cache, main-memory expansion, and optionally containing directory information. Each Mnemosyne device in turn supports up to four banks of DRAM, each 72 bits wide (64 bits + ECC). Using standard DRAM components, Terpsichore and Mnemosyne achieve bandwidth in excess of 9 Gbytes/sec to secondary cache and 2 Gbytes/sec to main memory. Terpsichore may use twice or four times the number of Mnemosyne devices to expand the cache and memory and to increase the bandwidth of the main memory system to in excess of 8 Gbytes/sec.

MU 0023410

Architecture Framework

The Mnemosyne architecture builds upon MicroUnity's Hermes high-bandwidth channel architecture and upon MicroUnity's Cerberus serial bus architecture, and complies with the requirements of Hermes and Cerberus. Mnemosyne uses parameters A and W as defined by Hermes.